



Licenciatura Engenharia Informática e Multimédia

Computação Física – CF

Relatório Trabalho Prático 1

Docente Carlos Carvalho

Trabalho realizado por:

Fábio Dias, nº 42921

Tatiana Cristão, nº 47508

Índice

Índice	2
Índice de Figuras	3
1. Exercício 1	7
Código	11
Montagem	13
Testes	20
2. Exercício 2	30
Código	46
Montagem	50
Testes	57
3. Exercício 3	67
Código	77
Montagem	82
Testes	91
4. Código	99
Exercício 1	99
Exercício 2	100
Exercício 3	102
5. Conclusões	104
6. Bibliografia	105

Índice de Figuras

Figura 1	7
Figura 2	8
Figura 3	9
Figura 4	9
Figura 5	10
Figura 6	11
Figura 7	11
Figura 8	11
Figura 9	11
Figura 10	11
Figura 11	12
Figura 12	12
Figura 13	12
Figura 14	13
Figura 15	14
Figura 16	15
Figura 17	16
Figura 18	17
Figura 19	18
Figura 20	19
Figura 21	20
Figura 22	21
Figura 23	22
Figura 24	23
Figura 25	24
Figura 26	25
Figura 27	26
Figura 28	27
Figura 29	28
Figura 30	29
Figura 31	31
Figura 32	32
Figura 33	32
Figura 34	33
Figura 35	33
Figura 36	34
Figura 37	34
Figura 38	35
Figura 39	36
Figura 40	37
Figura 41	38
Figura 42	38
Figura 43	38

Figura 44	38
Figura 45	39
Figura 46	39
Figura 47	39
Figura 48	39
Figura 49	39
Figura 50	39
Figura 51	40
Figura 52	40
Figura 53	41
Figura 54	41
Figura 55	42
Figura 56	42
Figura 57	43
Figura 58	43
Figura 59	44
Figura 60	44
Figura 61	44
Figura 62	45
Figura 63	46
Figura 64	46
Figura 65	46
Figura 66	47
Figura 67	47
Figura 68	47
Figura 69	47
Figura 70	48
Figura 71	48
Figura 72	49
Figura 73	50
Figura 74	51
Figura 75	52
Figura 76	53
Figura 77	54
Figura 78	55
Figura 79	56
Figura 80	57
Figura 81	58
Figura 82	59
Figura 83	60
Figura 84	61
Figura 85	62
Figura 86	63
Figura 87	64

Figura 88	65
Figura 89	66
Figura 90	68
Figura 91	68
Figura 92	69
Figura 93	70
Figura 94	71
Figura 95	72
Figura 96	73
Figura 97	74
Figura 98	75
Figura 99	75
Figura 100	75
Figura 101	75
Figura 102	75
Figura 103	75
Figura 104	76
Figura 105	76
Figura 106	77
Figura 107	77
Figura 108	77
Figura 109	77
Figura 110	78
Figura 111	78
Figura 112	78
Figura 113	78
Figura 114	79
Figura 115	79
Figura 116	80
Figura 117	80
Figura 118	81
Figura 119	81
Figura 120	82
Figura 121	83
Figura 122	84
Figura 123	85
Figura 124	86
Figura 125	87
Figura 126	88
Figura 127	89
Figura 128	90
Figura 129	91
Figura 130	92
Figura 131	93

Figura 132	94
Figura 133	95
Figura 134	96
Figura 135	97
Figura 136	98

1. Exercício 1

Para este exercício, foi-nos pedido para realizar um pequeno programa que detecta o mal funcionamento das portas lógicas AND e OR. Caso um comportamento anómalo se verifique, é acendido um LED.

Dado que vamos ter de verificar duas portas lógicas, decidimos ter um selector e definimos que, quando o selector toma o valor 0, estamos a verificar a porta AND; quando o selector toma o valor 1, estamos a verificar a porta OR.

O primeiro passo é definir o modelo geral, onde identificamos as entradas e saídas deste circuito. (*Ver Figura 1*)

Como as tabelas das portas lógicas já são conhecidas^[1], conseguimos prever qual o comportamento expectável. Se este comportamento não se revelar, temos um comportamento anómalo, logo, o LED tem de ser acesso.

Comecemos por definir uma tabela de verdade deste programa. As variáveis de entrada são S, de Selector, F, de Função, ou seja, o resultado das portas lógicas, A e B, duas variáveis de entrada das portas lógicas. Como saída, temos L, de LED. (*Ver Figura 2*)

A partir desta tabela de verdade, conseguimos obter o mapa de Karnaugh^[1], que nos permite simplificar a expressão, agrupando os “1”s que são adjacentes ou simétricos, reduzindo assim uma variável. (*Ver Figura 3 e Figura 4*)

Adicionalmente, com a expressão de L, conseguimos montar um circuito lógico que permite o correcto funcionamento deste programa. (*Ver Figura 5*)

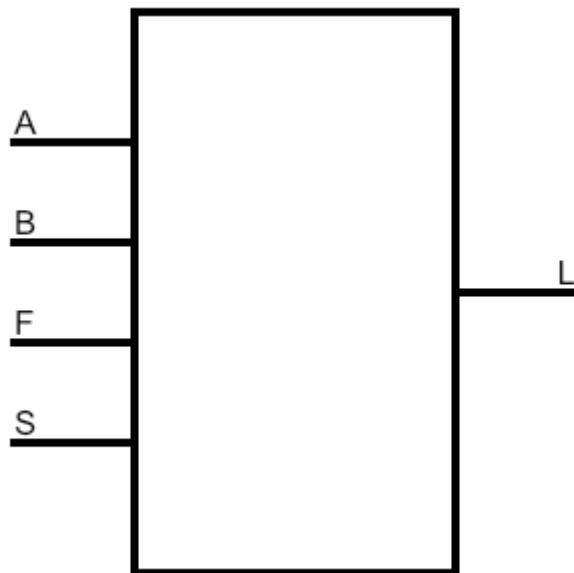


Figura 1 - Modelo Geral

S	F	A	B	L
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figura 2 - Tabela de Verdade de L

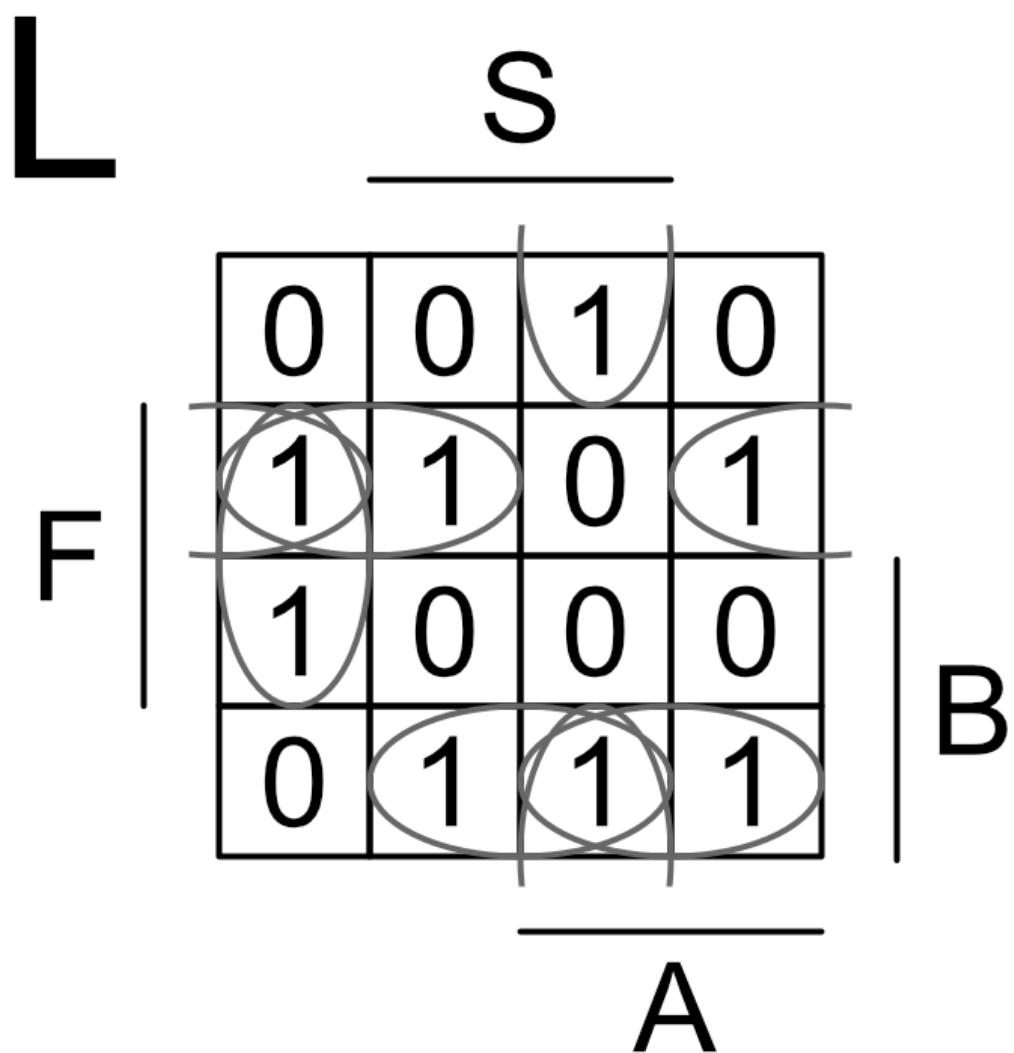


Figura 3 - Mapa de Karnaugh de L

$$L = S \cdot \bar{F} \cdot A + F \cdot \bar{A} \cdot \bar{B} + \bar{S} \cdot F \cdot \bar{B} + \bar{S} \cdot F \cdot \bar{A} + S \cdot \bar{F} \cdot B + \bar{F} \cdot A \cdot B$$

Figura 4 - Expressão de Lx

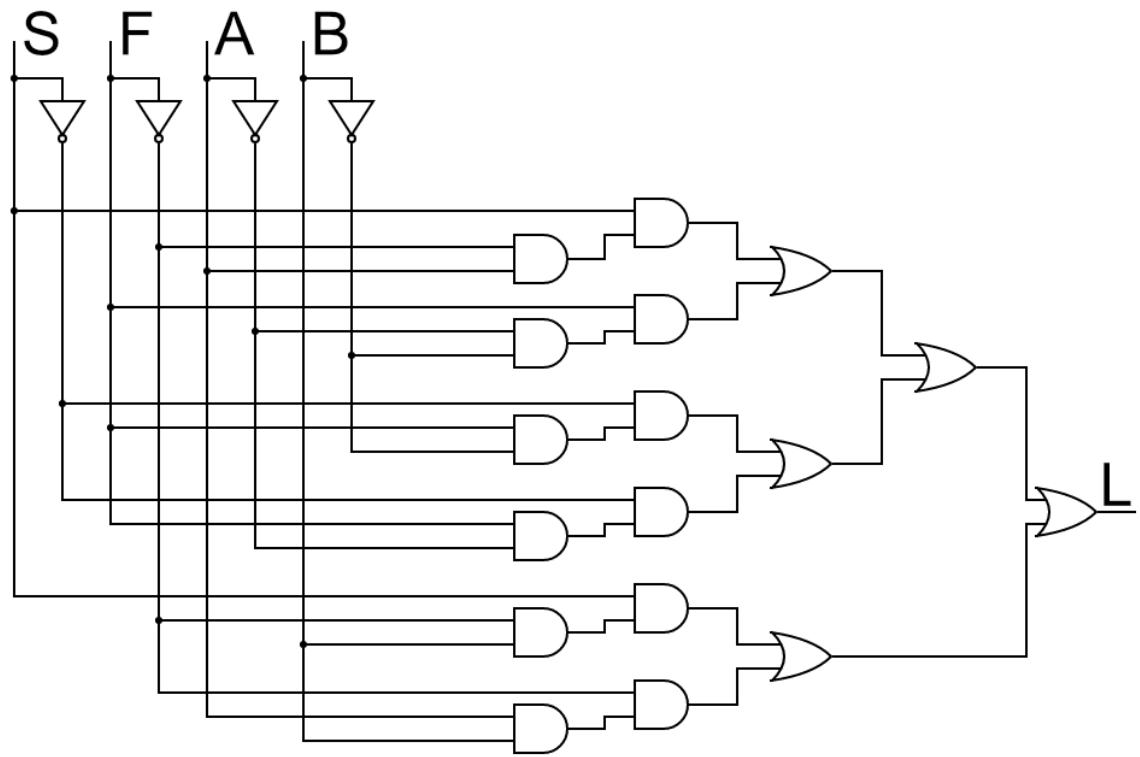


Figura 5 - Circuito Lógico

Código

Começamos por definir os pinos para cada uma das variáveis de entrada, A, B, F e S. E também o pino para a variável de saída, L. (*Ver Figura 6*)

```
#define pinA 2
#define pinB 3
#define pinF 4
#define pinS 5
#define pinL 6
```

Figura 6 - Definição de Pinos

Definimos as variáveis a serem usadas, neste caso dado que só podem tomar valores de 0 ou 1, estas serão booleanas. (*Ver Figura 7*)

```
boolean A, B, F, S, L;
```

Figura 7 - Definição de Variáveis

É implementada a expressão do LED, obtida previamente. (*Ver Figura 8*)

```
boolean Led(boolean A, boolean B, boolean F, boolean S)
{
    return S & !F & A | F & !A & !B | !S & F & !B | !S & F & !A | S & !F & B | !F & A & B;
}
```

Figura 8 - Implementação de Led

Também implementamos uma função para ler os inputs, ou seja, para obter os valores correntes das variáveis de entrada. (*Ver Figura 9*)

```
void readInputs()
{
    A = digitalRead(pinA);
    B = digitalRead(pinB);
    F = digitalRead(pinF);
    S = digitalRead(pinS);
}
```

Figura 9 - Implementação de readInputs

Tal como foi feito para os inputs, também tem de ser feito para o output, ou seja, para a variável de saída. (*Ver Figura 10*)

```
void writeOutputs()
{
    digitalWrite(pinL, L);
}
```

Figura 10 - Implementação de writeOutputs

Dado que a variável L é a que vai demonstrar se existe um comportamento anómalo, temos de a afetar com a expressão arranjada. (*Ver Figura 11*)

```
void detectAnomaly()
{
    L = Led(A, B, F, S);
}
```

Figura 11 - Implementação de detectAnomaly

No *setup* temos de definir o modo dos pinos previamente definidos. (*Ver Figura 12*)

```
void setup() {
    pinMode(pinA, INPUT);
    pinMode(pinB, INPUT);
    pinMode(pinF, INPUT);
    pinMode(pinS, INPUT);
    pinMode(pinL, OUTPUT);
}
```

Figura 12 - Setup

Por fim, temos o *loop* do programa. Neste, temos apenas de ler os valores de entrada, afectar L e enviar o seu valor para o LED. (*Ver Figura 13*)

```
void loop() {
    readInputs();
    detectAnomaly();
    writeOutputs();
}
```

Figura 13 - Loop

Montagem

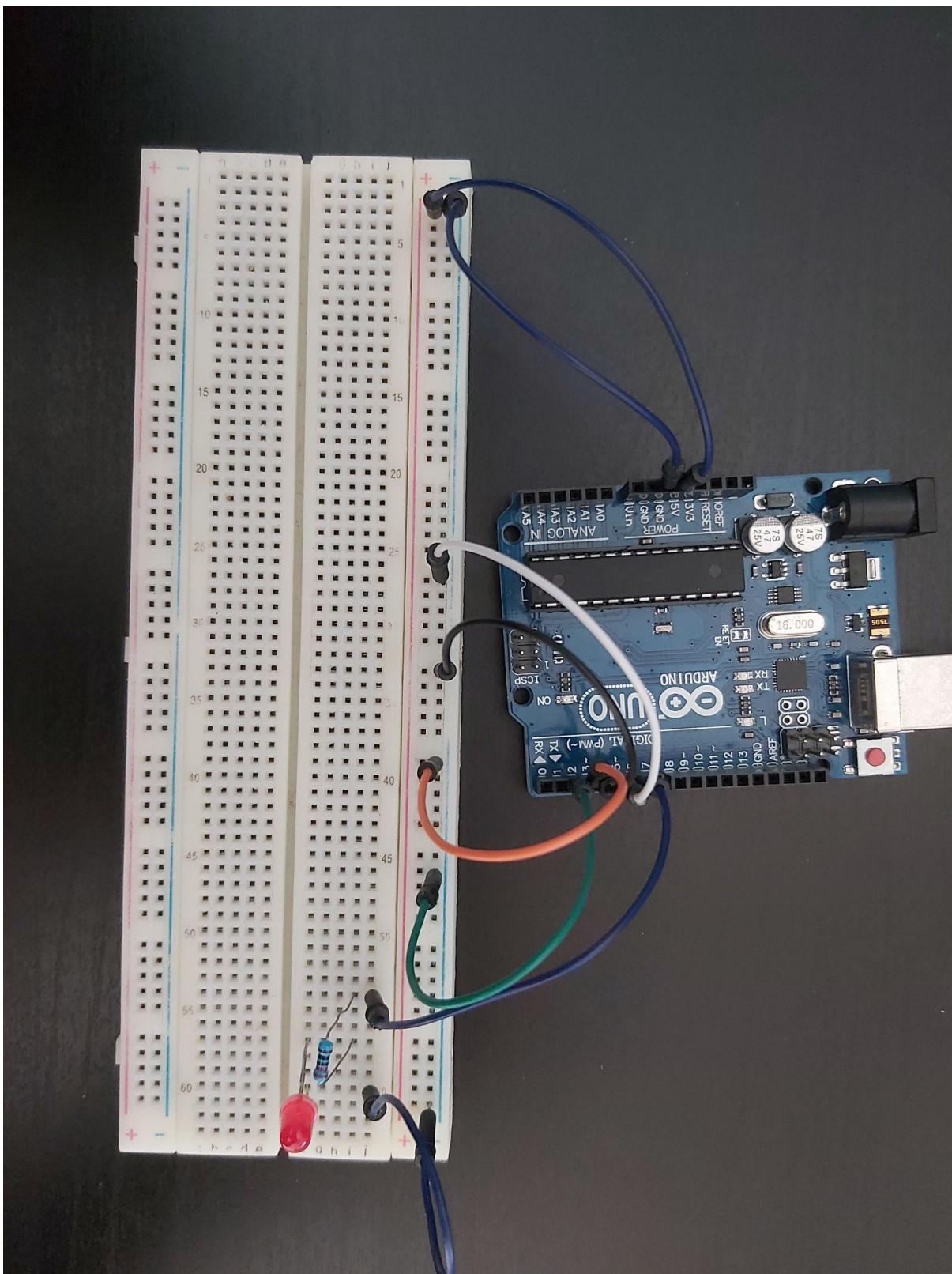


Figura 14 - Montagem Geral

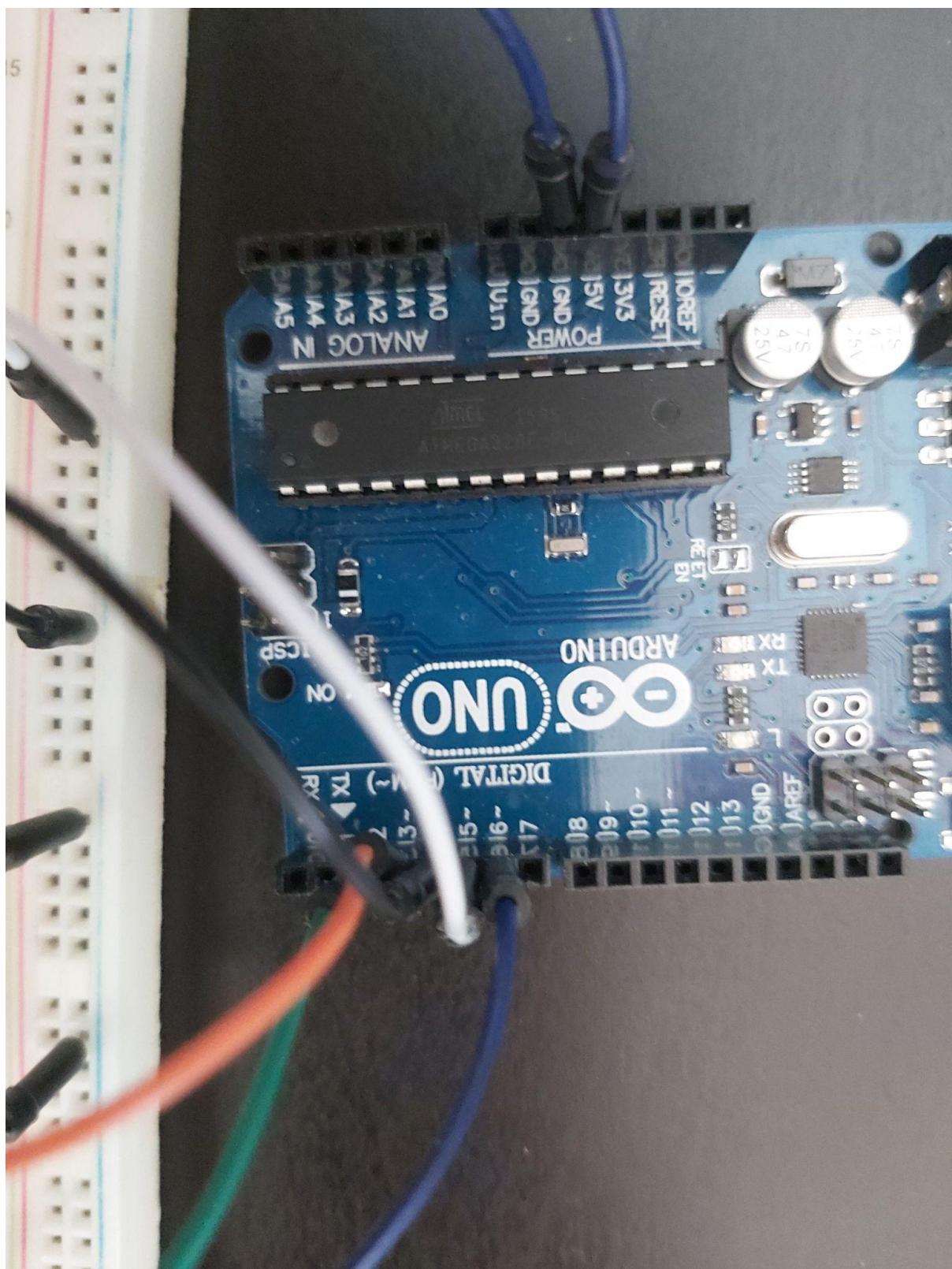


Figura 15 - Montagem Arduíno 1

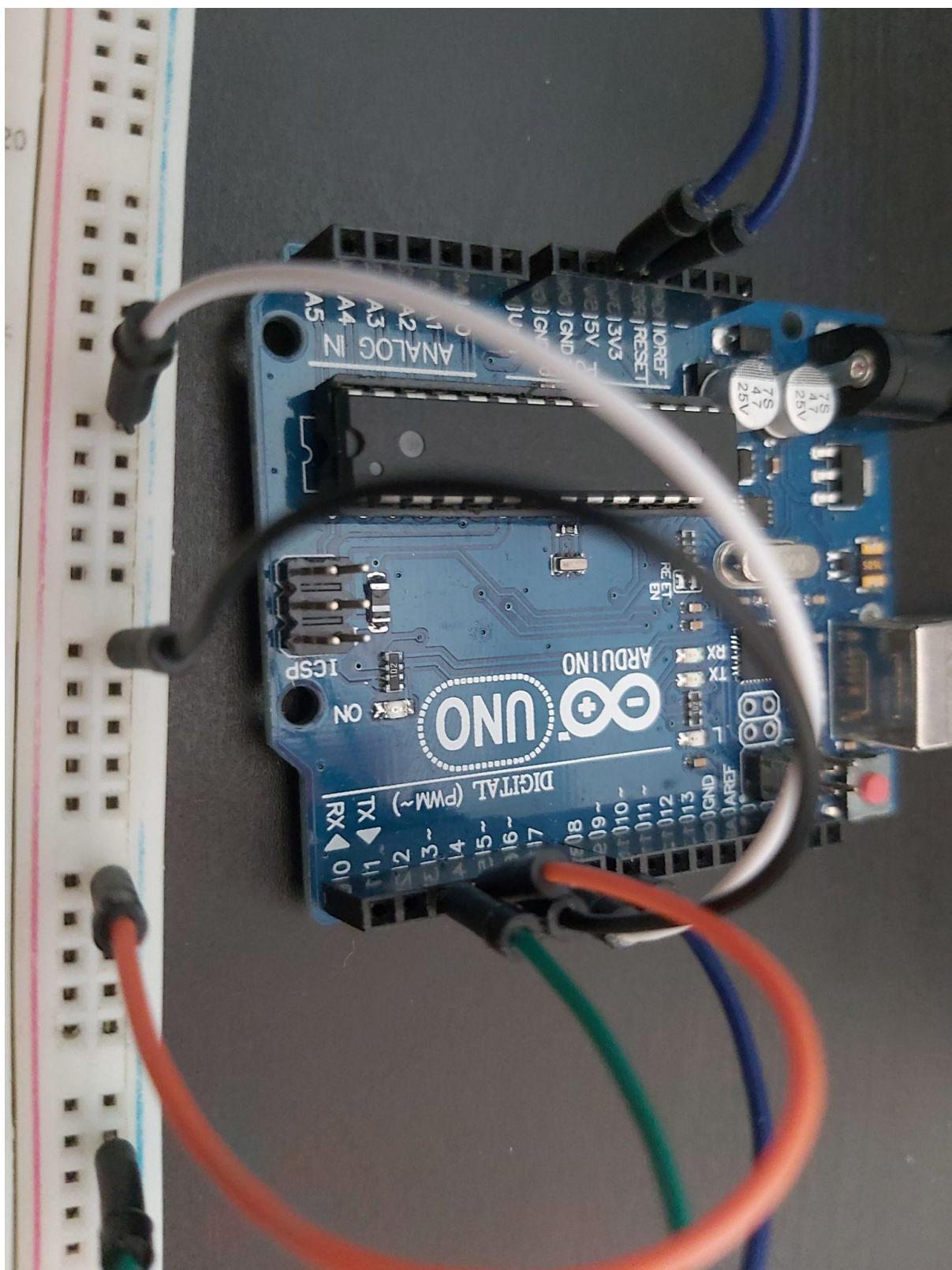


Figura 16 - Montagem Arduíno 2

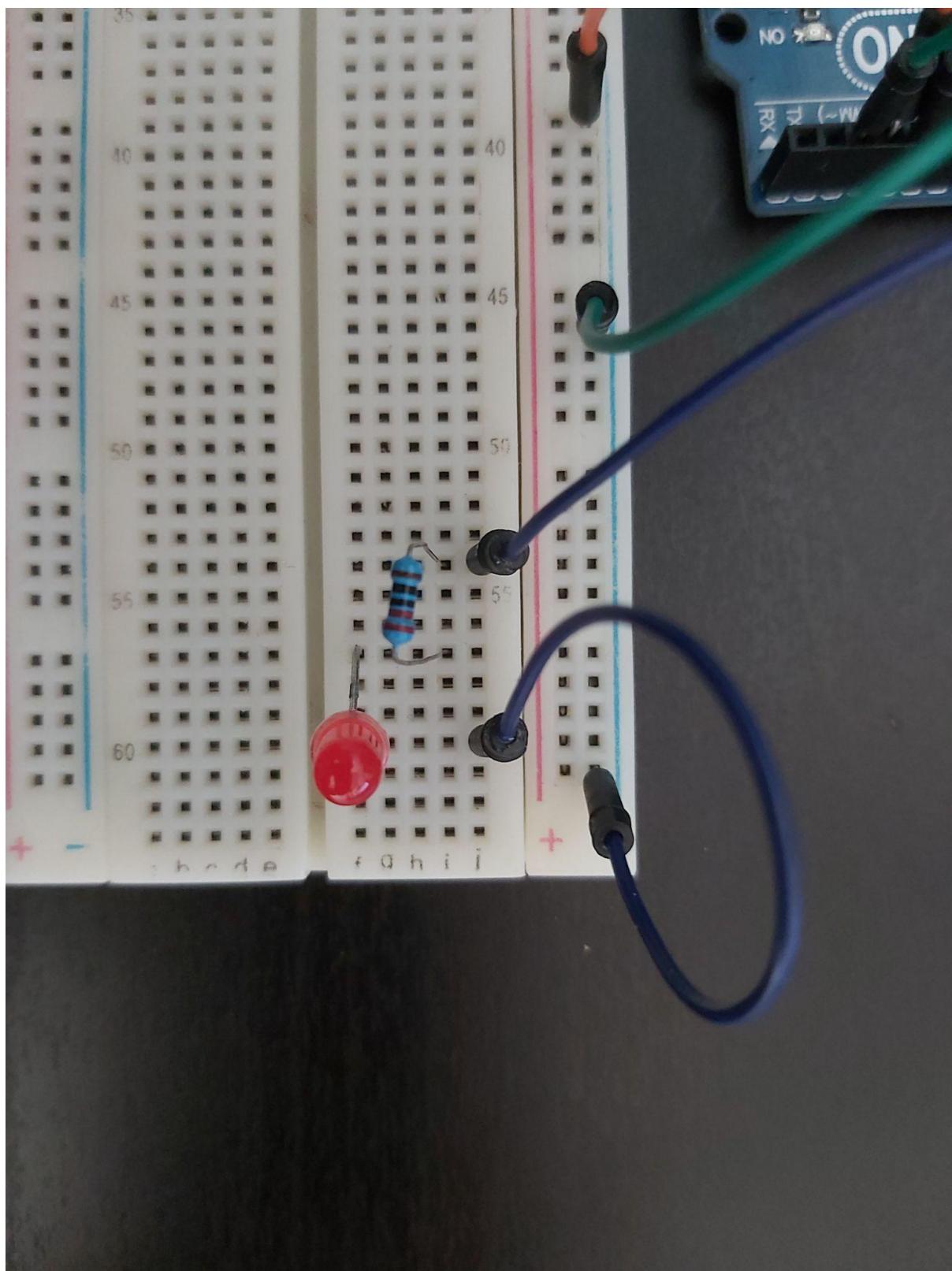


Figura 17 - Montagem LED 1

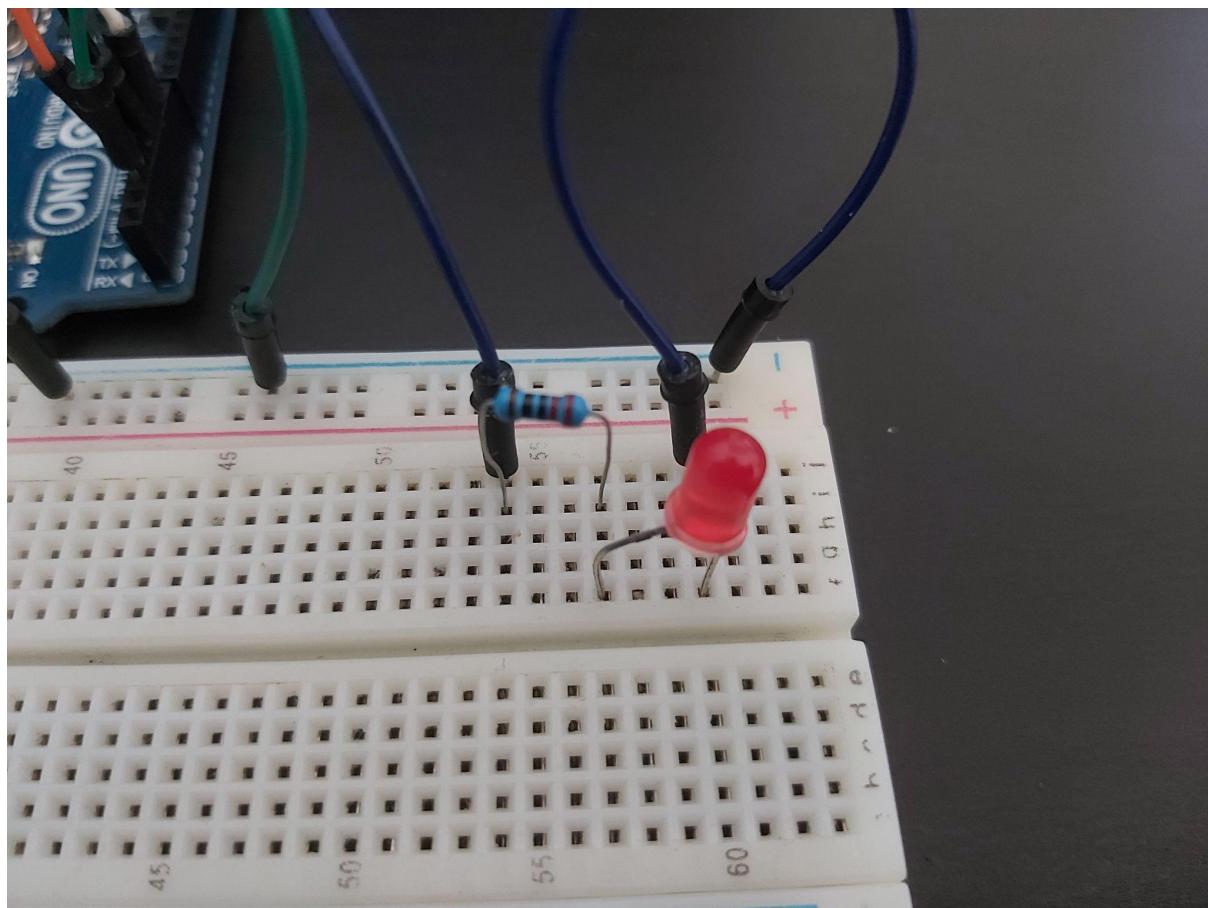


Figura 18 - Montagem LED 2

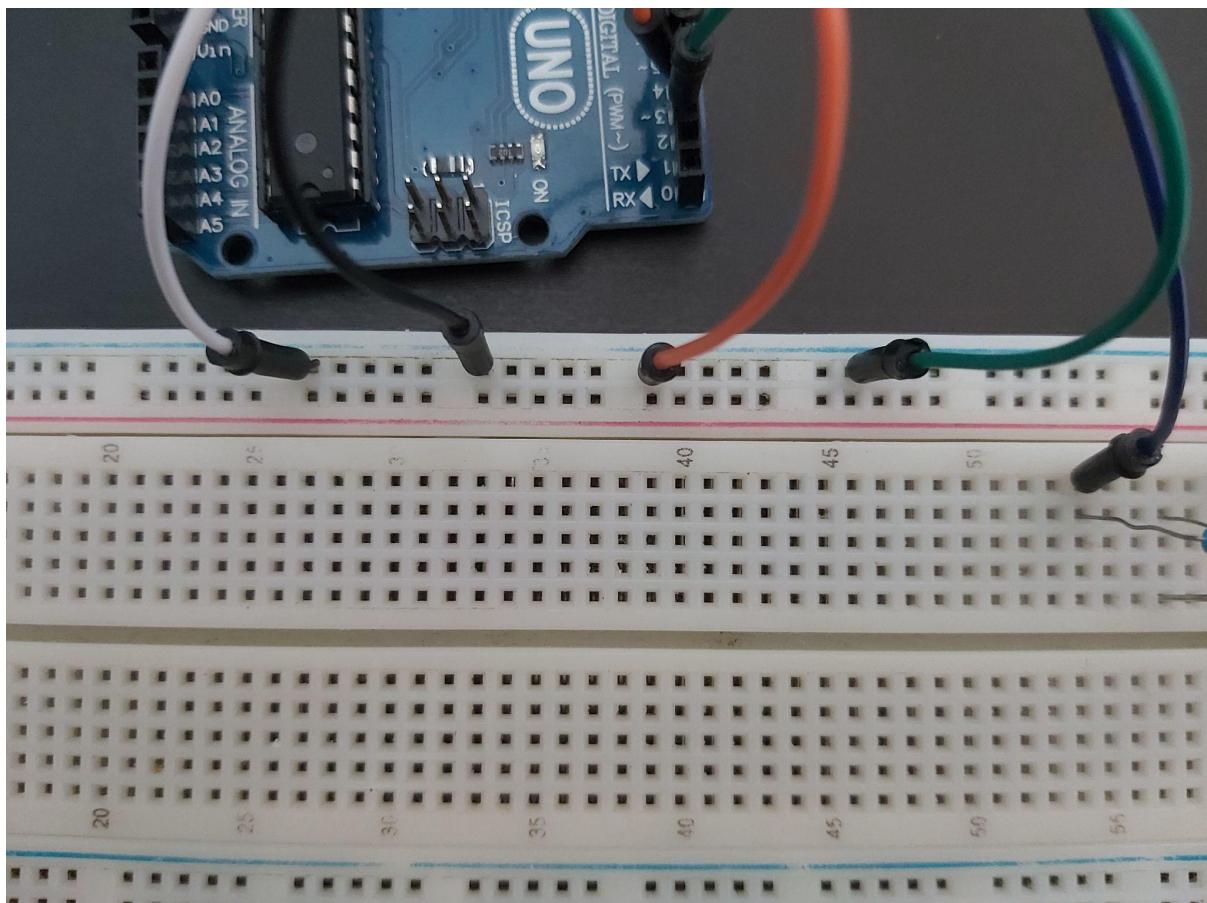


Figura 19 - Montagem Valores Pinos 1

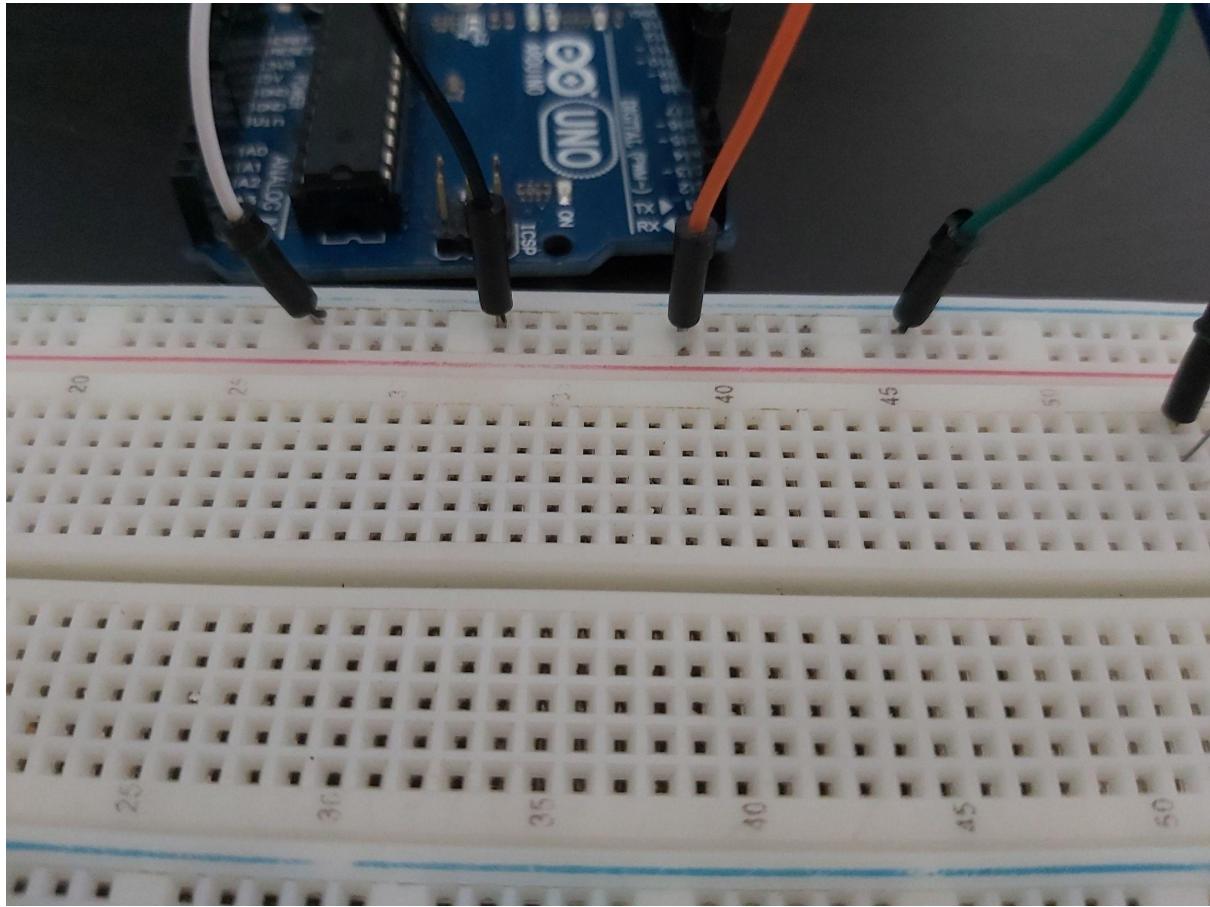


Figura 20 - Montagem Valores de Pinos 2

Testes

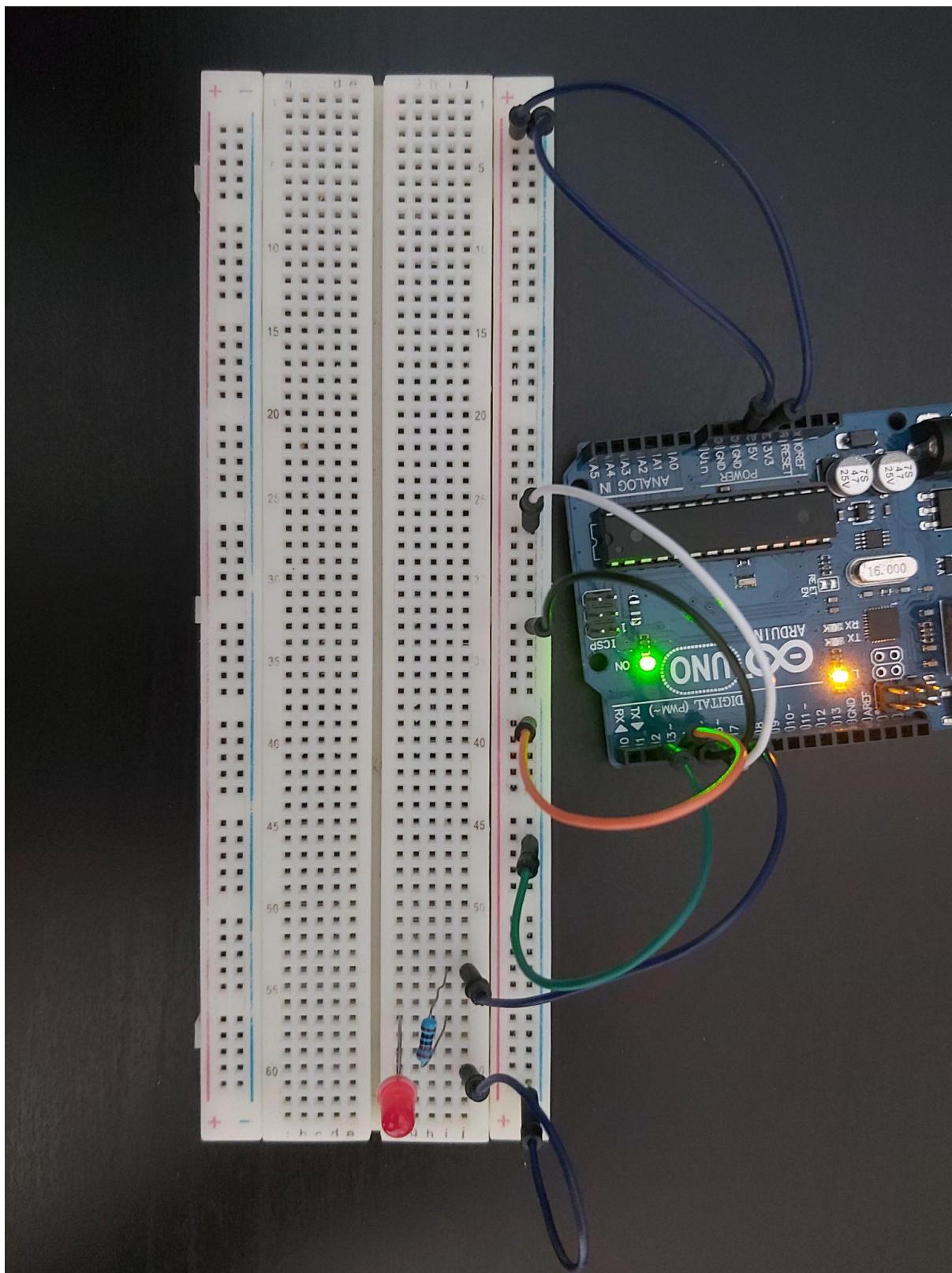


Figura 21 - Testes AND: A = 0, B = 0, F = 0

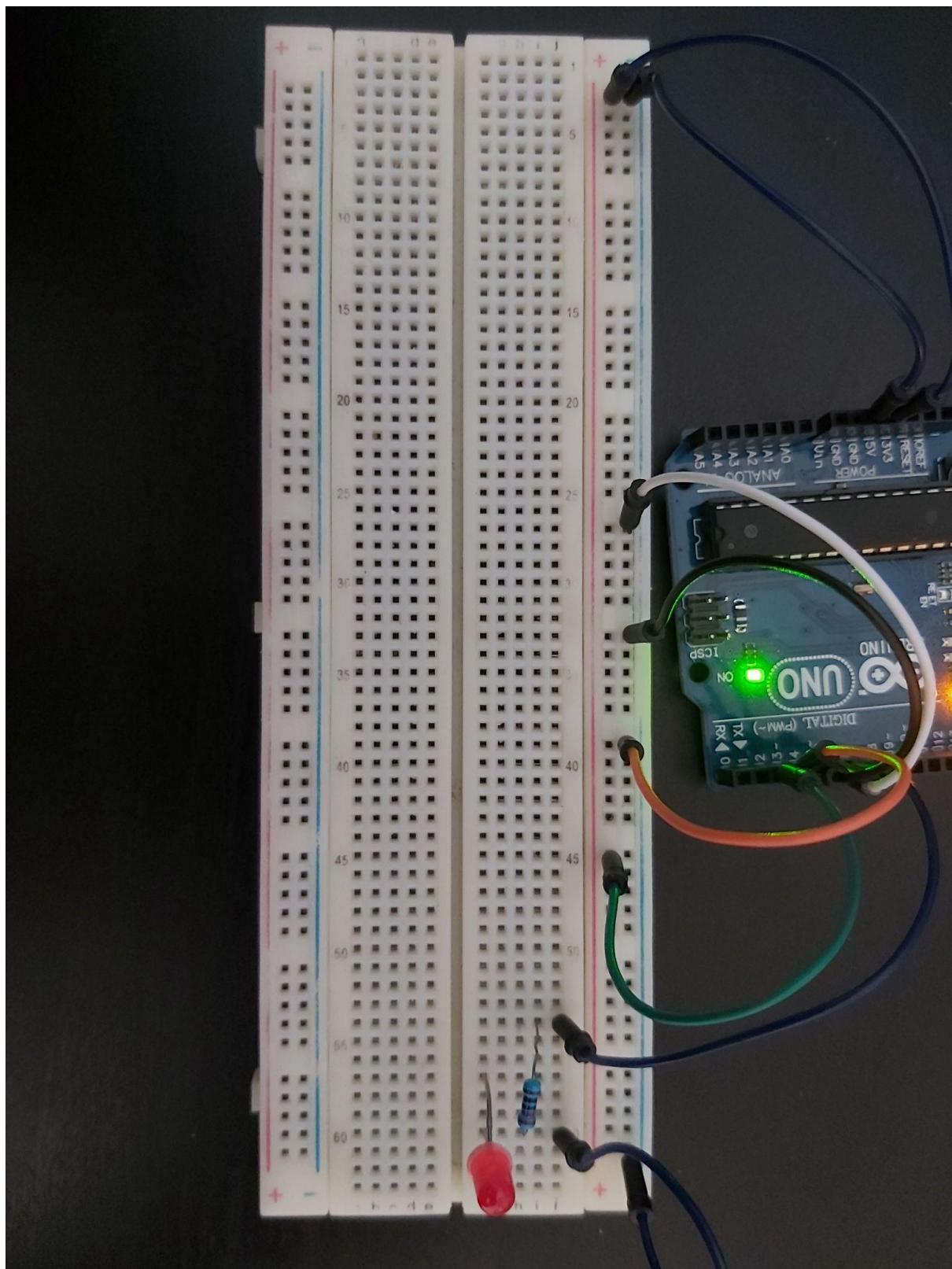


Figura 22 - Testes AND: A = 1, B = 0, F = 0

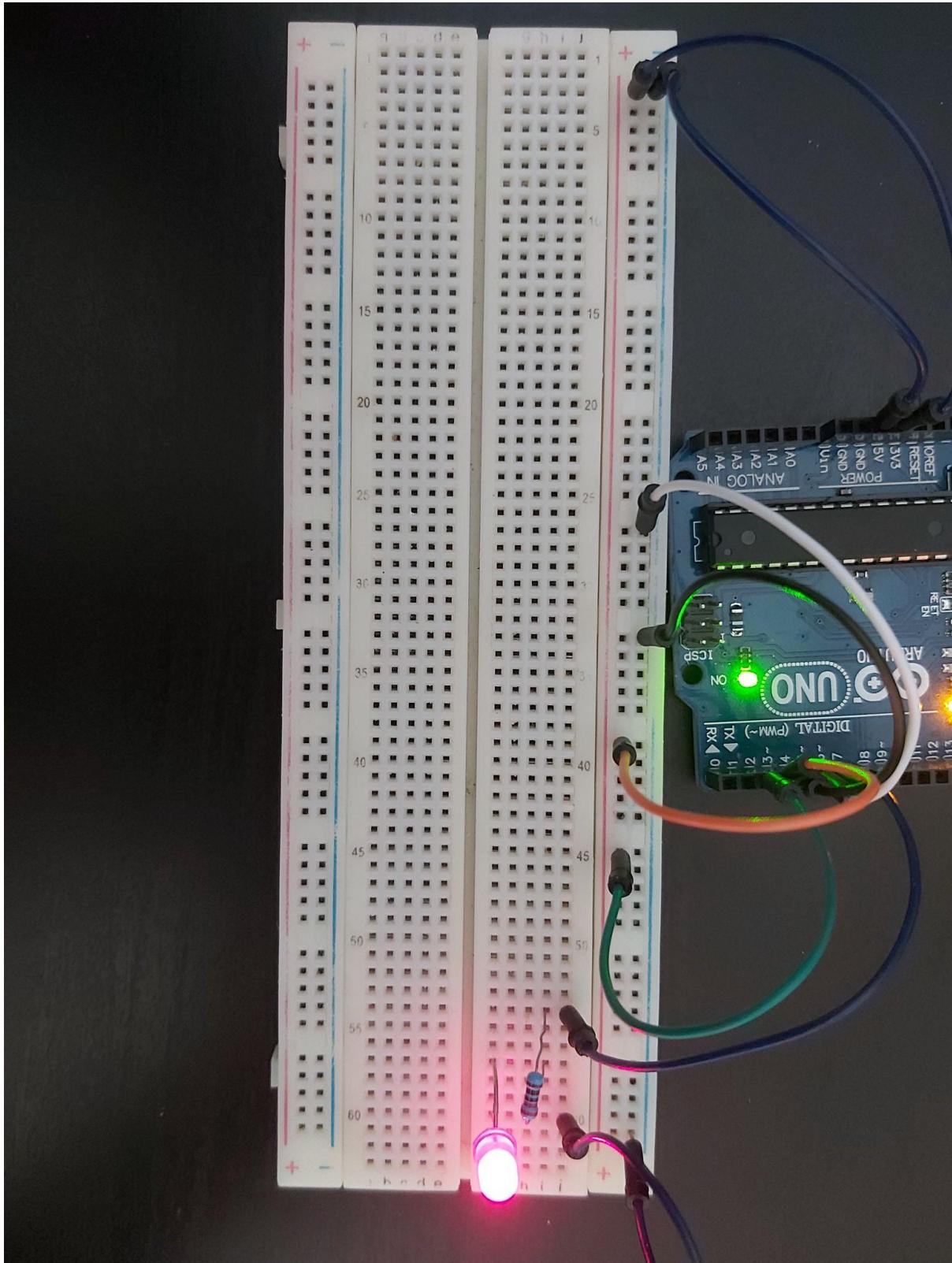


Figura 23 - Testes AND: A = 1, B = 1, F = 0

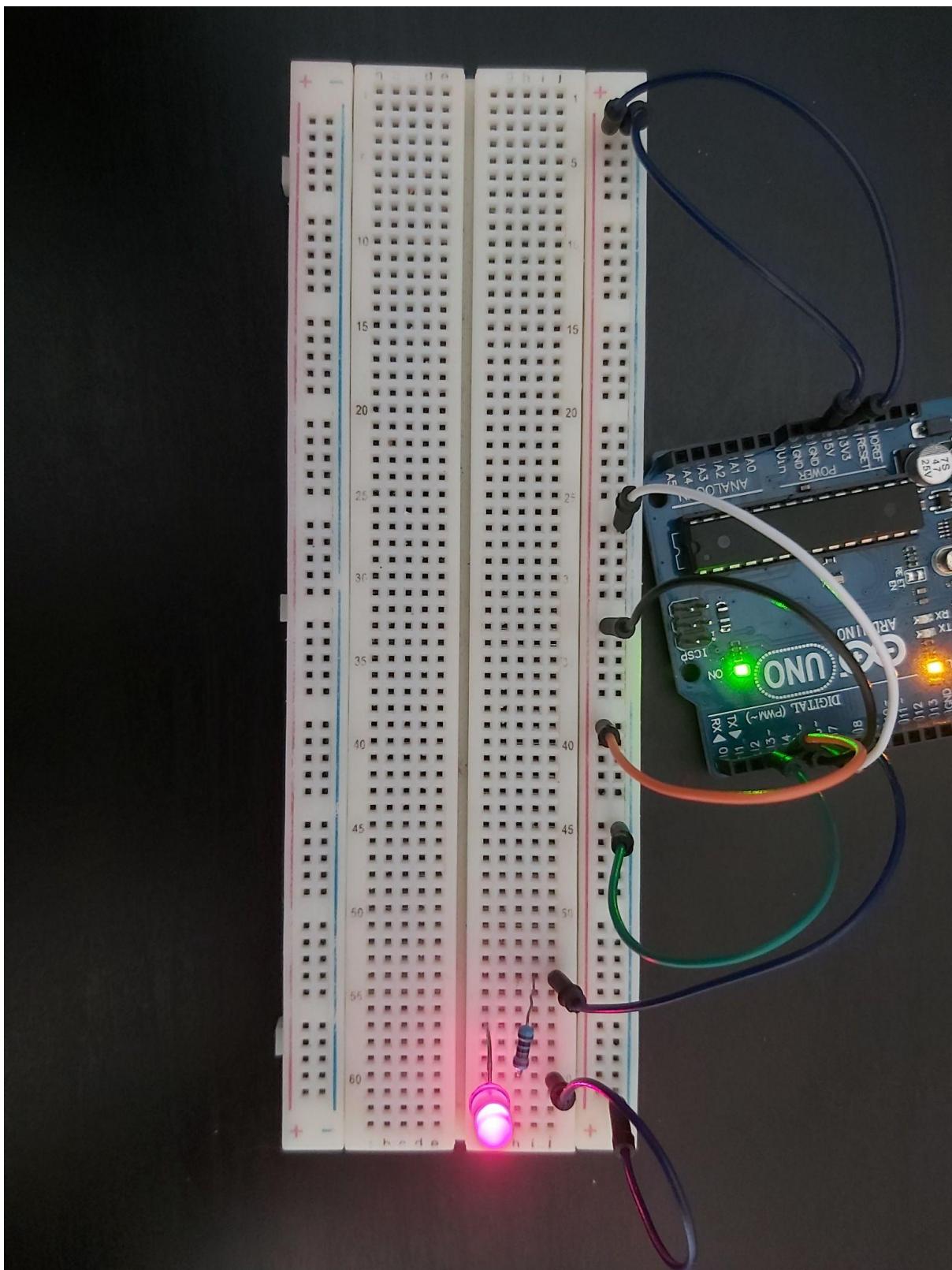


Figura 24 - Testes AND: A = 0, B = 1, F = 1

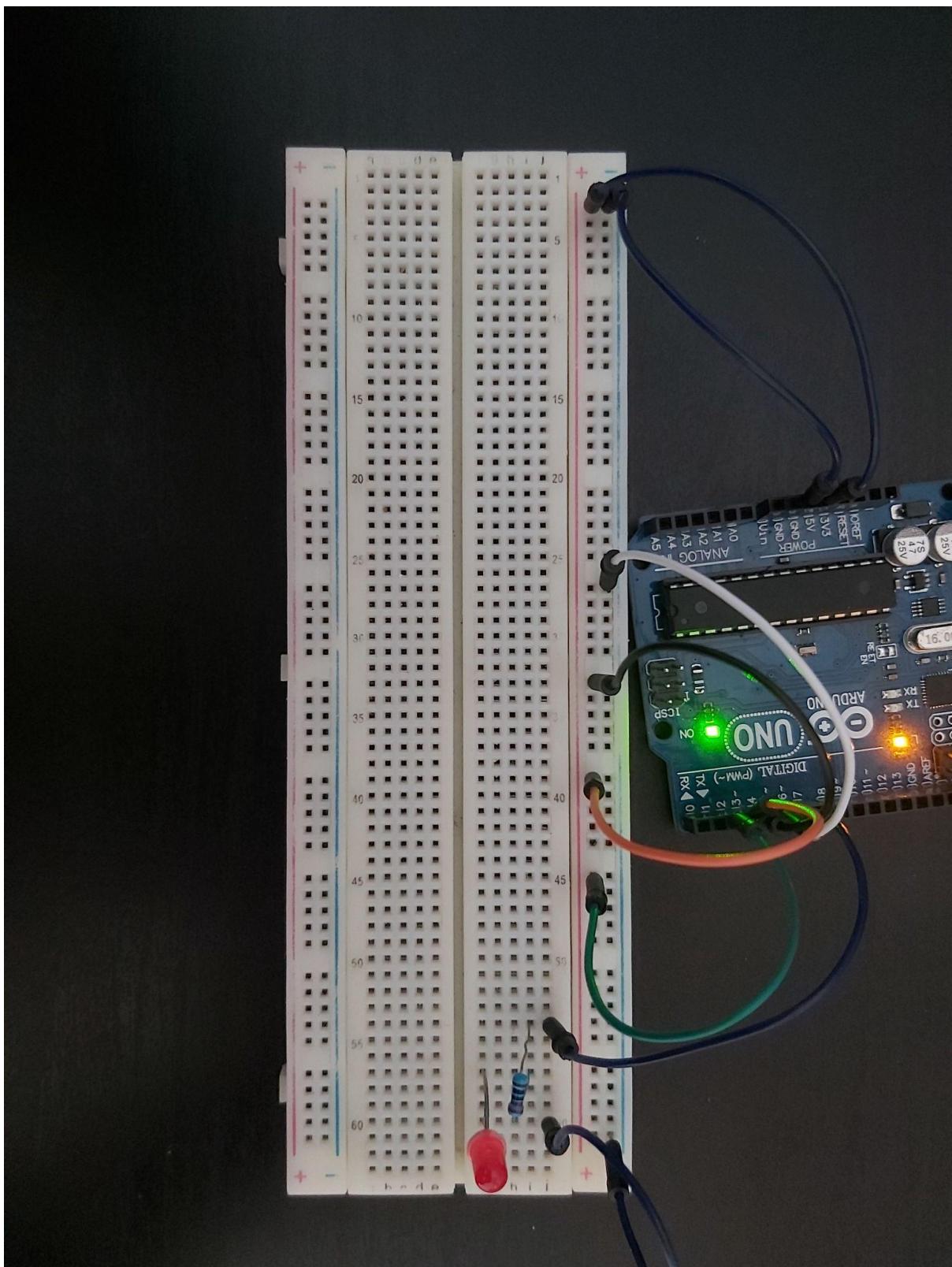


Figura 25 - Testes AND: A = 1, B = 1, F = 1

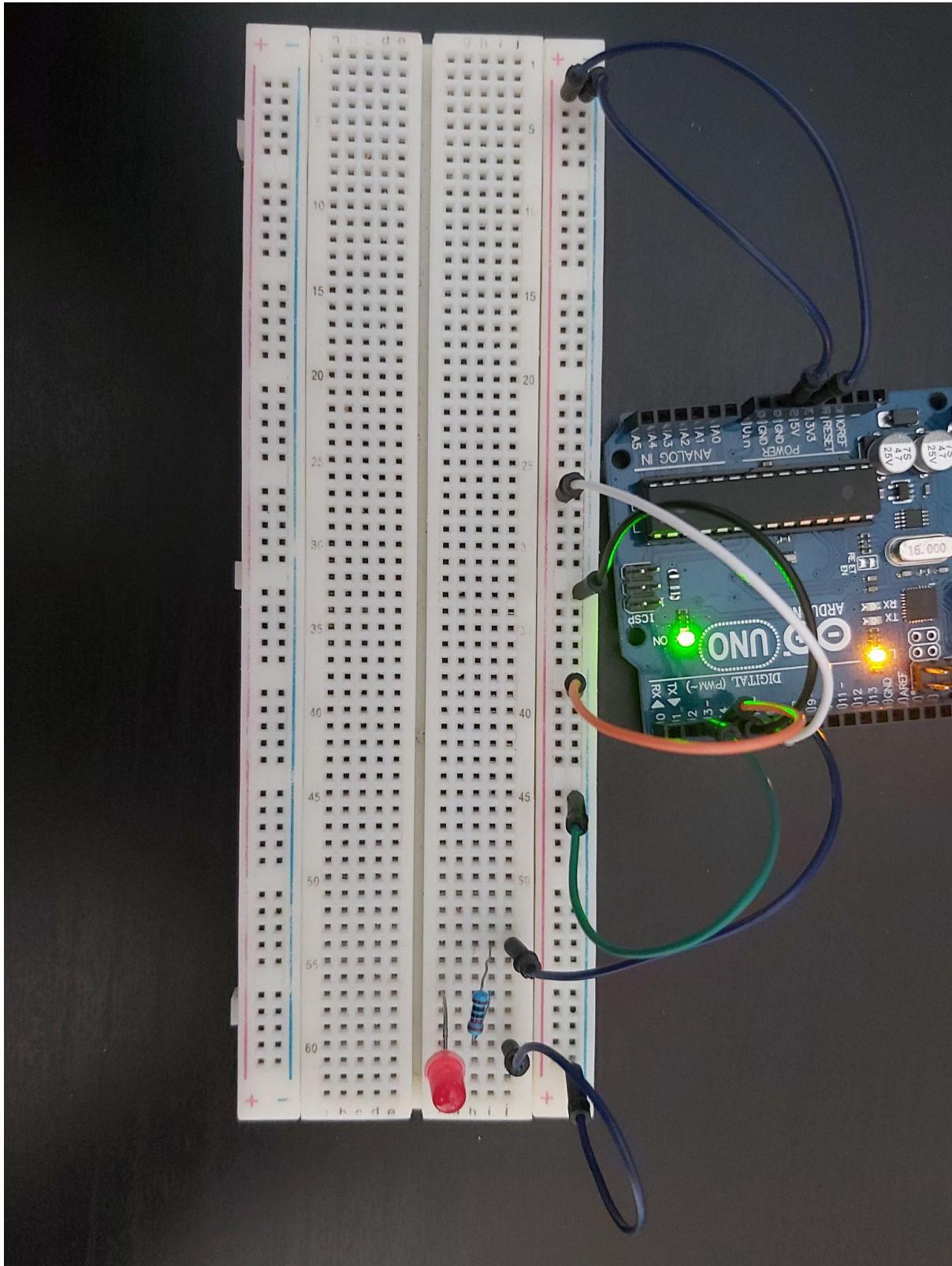


Figura 26 - Testes OR: A = 0, B = 0, F = 0

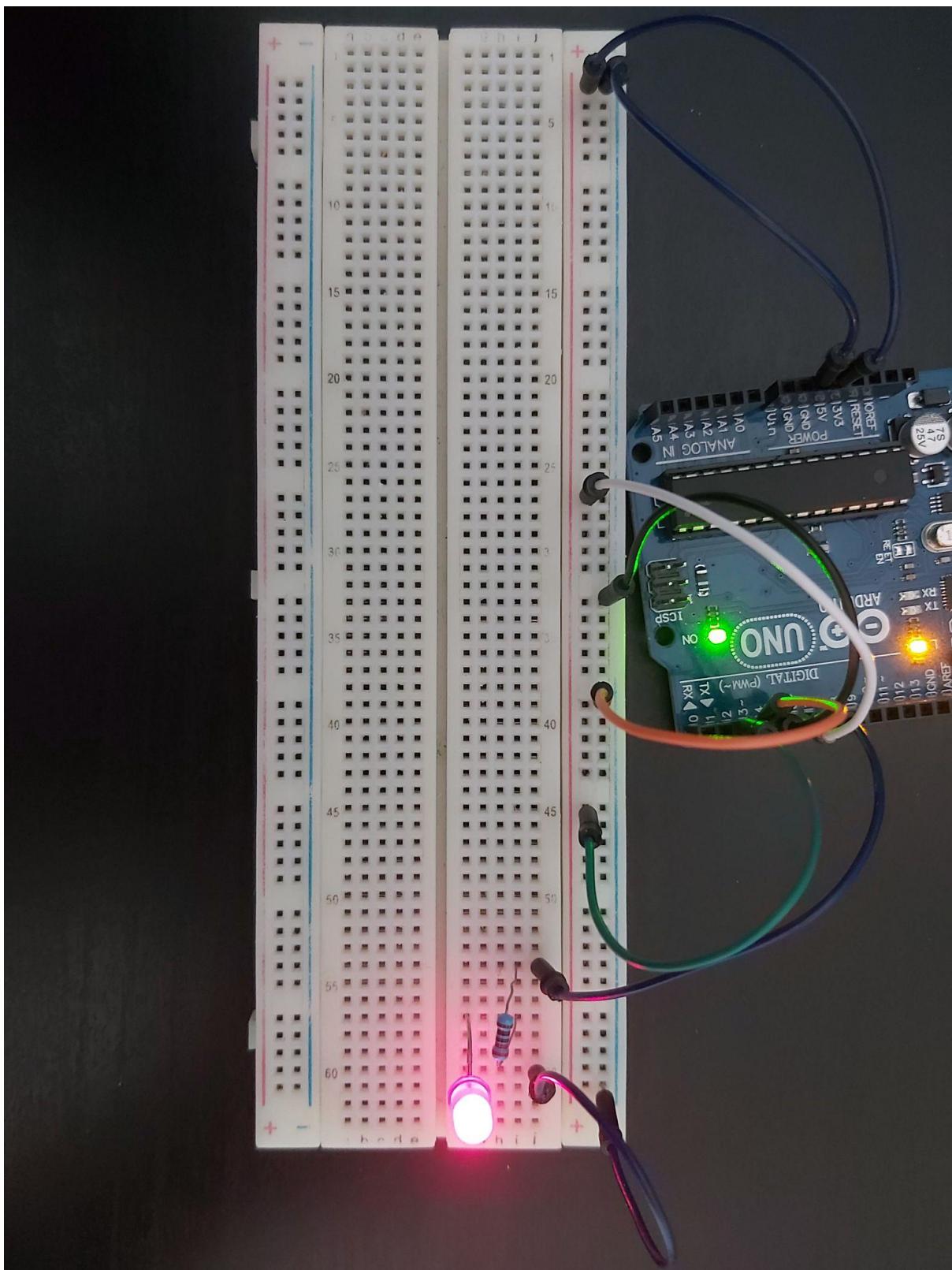


Figura 27 - Testes OR: A = 1, B = 0, F = 0

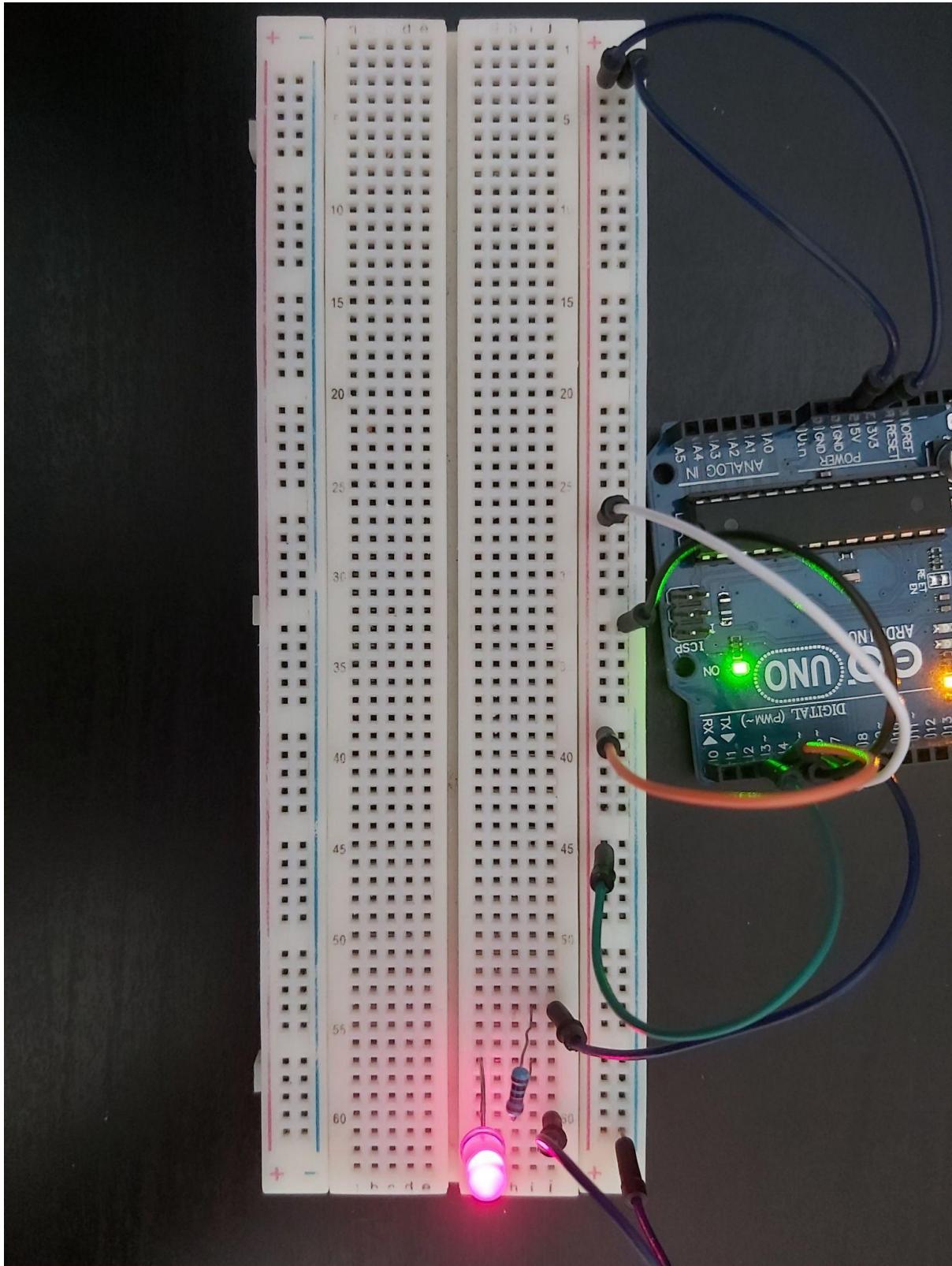


Figura 28 - Testes OR: A = 1, B = 1, F = 0

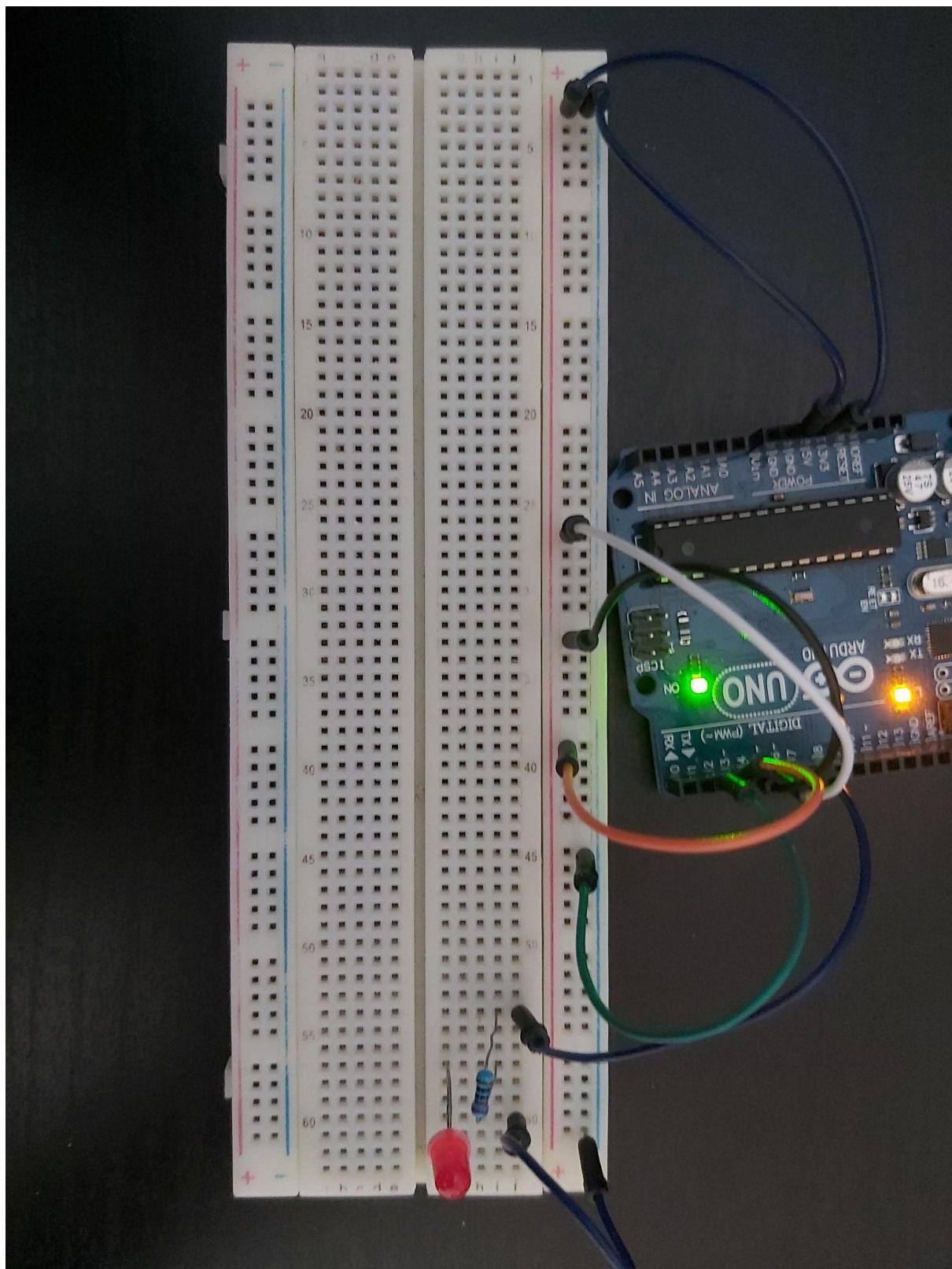


Figura 29 - Testes OR: A = 0, B = 1, F = 1

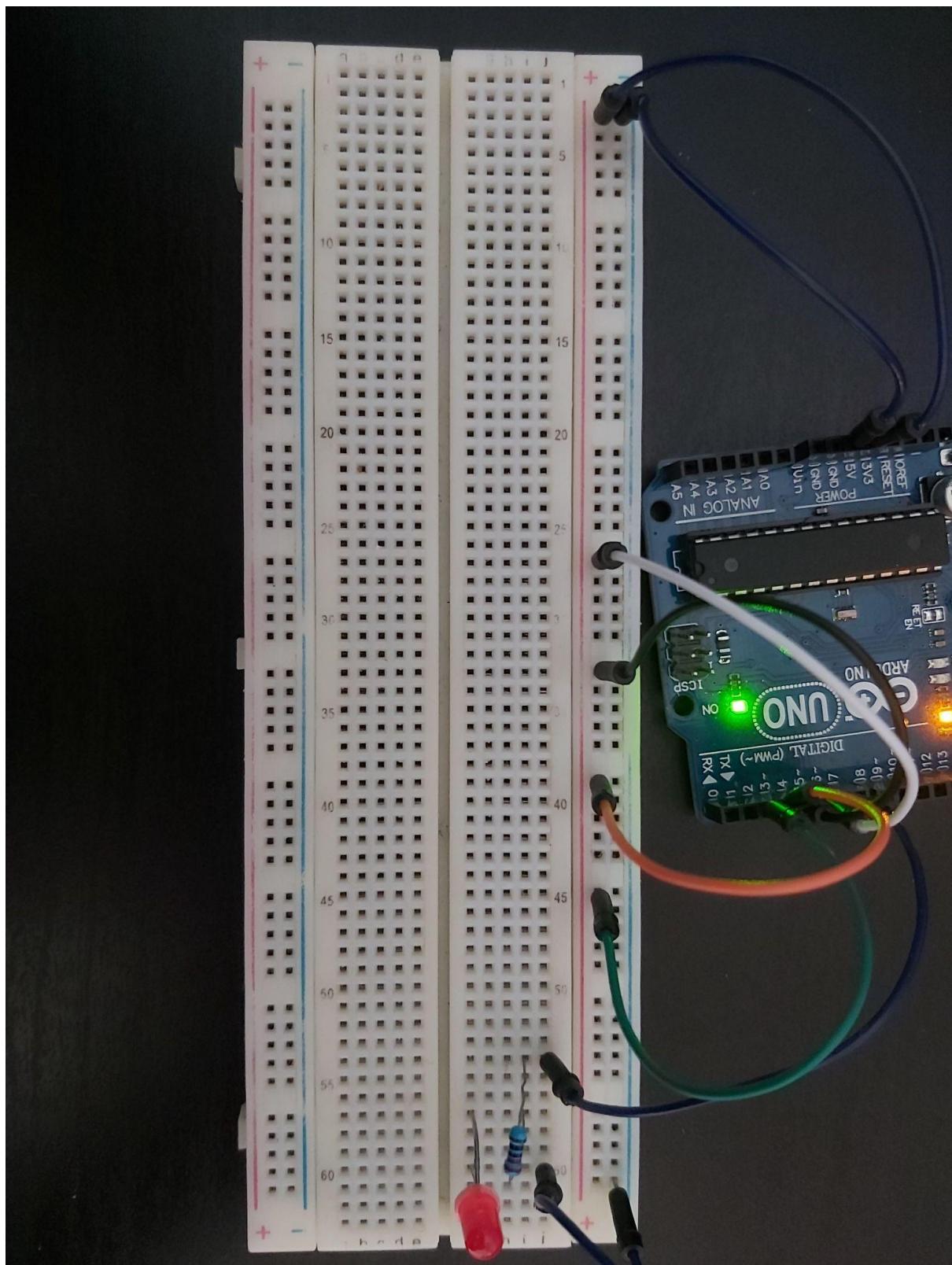


Figura 30 - Testes OR: A = 1, B = 1, F = 1

2. Exercício 2

Para este segundo exercício, foi-nos pedido a projecção de um somador e subtrator, de dois número binários, ambos a três bits. E obter um resultado também a três bits.

Para além disto, são necessárias as *flags*: *Carry* (*Cy*) e *Borrow* (*Bw*), *Overflow* (*Overflow*) e ainda *Zero* (*Zero*).

Dado que são duas operações diferentes, será preciso um selector e definimos que, quando toma o valor 0, estamos a executar a soma; quando o selector toma o valor 1, estamos a executar a subtração. Com isto, obtemos o modelo geral. (Ver Figura 31)

Visto que existem três bits para cada número e para a saída, temos de tomar uma abordagem modular^[1]. Ou seja, um caso geral que possa ser aplicado a cada bit singular mas, a junção desta projecta o resultado correto e esperado. Para tal, temos de usar os módulos de somador completo e os de subtrator completo^[1]. Ou seja, temos de realizar a tabela, tanto da soma como da subtração, e ter em conta o *carry* do caso passado, também chamado de *carry in* ou *carryN-1* e o *borrow* passado, também chamado de *borrow in* ou *borrowN-1*, para cada caso, respectivamente. Com estas tabelas, pretendemos obter o *Sn*, o valor que o bit *N* da saída toma, assim como o *carry* actual, ou seja, o *carryN*, para o caso da soma e o *borrow* actual, ou seja, o *borrowN*. (Ver Figura 32, Figura 33, Figura 34 e Figura 35, Figura 36 e Figura 37)

Com estas tabelas de verdade, é possível obterem-se os mapas de Karnaugh para o *Sn* de cada operação, assim como o *carryN* e o *borrowN*. (Ver Figura 38, Figura 39, Figura 40 e Figura 41)

Desta forma, é possível obter as expressões lógicas mais simplificadas para cada caso. (Ver Figura 42, Figura 43, Figura 44 e Figura 45)

Como é possível observar, existem semelhanças entre as expressões lógicas do *carryN* e do *borrowN*. A única variável que realmente é diferente da expressão do *carryN* para a do *borrowN* é a *An*, em que se encontra complementada na expressão do *borrowN* e encontra-se não complementada na de *carryN*.

Como mencionado previamente, teremos um selector que indicará qual a operação a ser executada. Dado que existe esta semelhança entre o *carryN* e o *borrowN*, podemos utilizar o selector, conjugado com o *An* e, assim, obtermos, caso o selector seja 0, o *carryN* e, caso seja 1, o *borrowN*. A partir de agora, consideremos o *carryN* e o *borrowN* unidos e chamamos-lhe *CyBwN* para o bit atual e *CyBwN-1* para o bit anterior. Conseguimos construir uma tabela de verdade com o que pretendemos alcançar. (Ver Figura 46)

Como é possível observar, o comportamento descrito na tabela de verdade é o comportamento do XOR^[1]. Também é possível observar que a expressão lógica de *Sn* é igual para ambas as operações. (Ver Figura 47)

Dado que o número variáveis não complementares na expressão *Sn* é sempre ímpar, podemos simplificar esta expressão para um XOR entre as três variáveis. São assim obtidas as expressões finais para o *Sn* e para o *CyBw*. (Ver Figura 48 e Figura 49)

Não nos podemos esquecer que necessitamos de obter expressões para as flags de *Overflow* e *Zero*. Para obtermos o valor de *Overflow*, temos de aplicar um XOR entre $CyBwN$ e o $CyBwnN-1$. Para a flag *Zero*, dado que esta só é activada quando todos os valores de S_n são 0, é o comportamento exacto da porta lógica NOR^[1]. (Ver Figura 50, Figura 51 e Figura 52)

Para cada uma destas expressões lógicas, é possível montar todos estes circuitos lógicos. (Ver Figura 53, Figura 54, Figura 55, Figura 56, Figura 57, Figura 58, Figura 59, Figura 60 e Figura 61)

Depois deste plano todo concretizado, também podemos organizar um símbolo lógico. (Ver Figura 62)

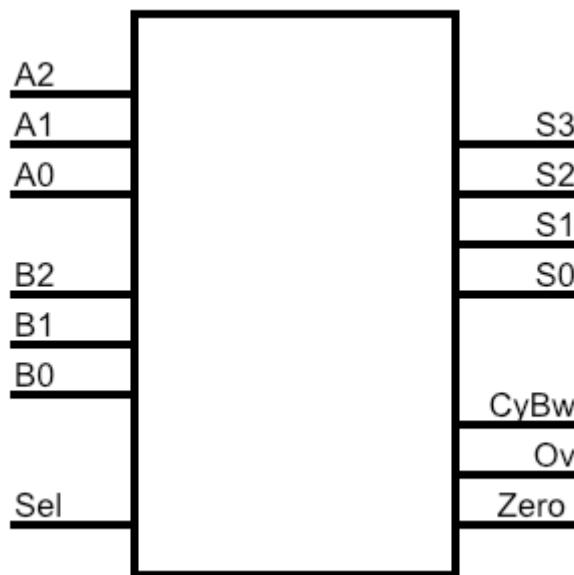


Figura 31 - Modelo Geral

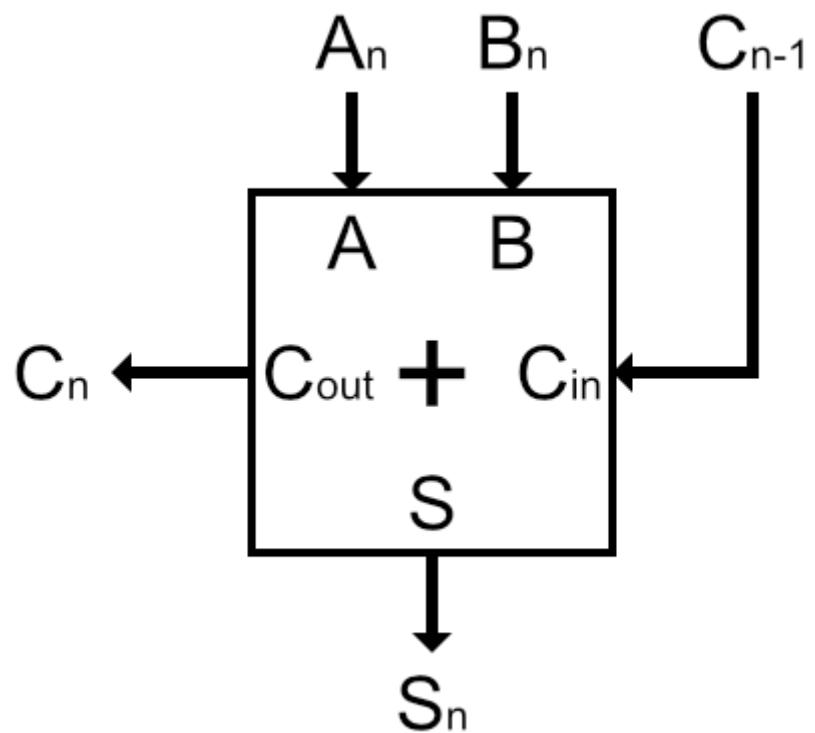


Figura 32 - Módulo Somador Completo

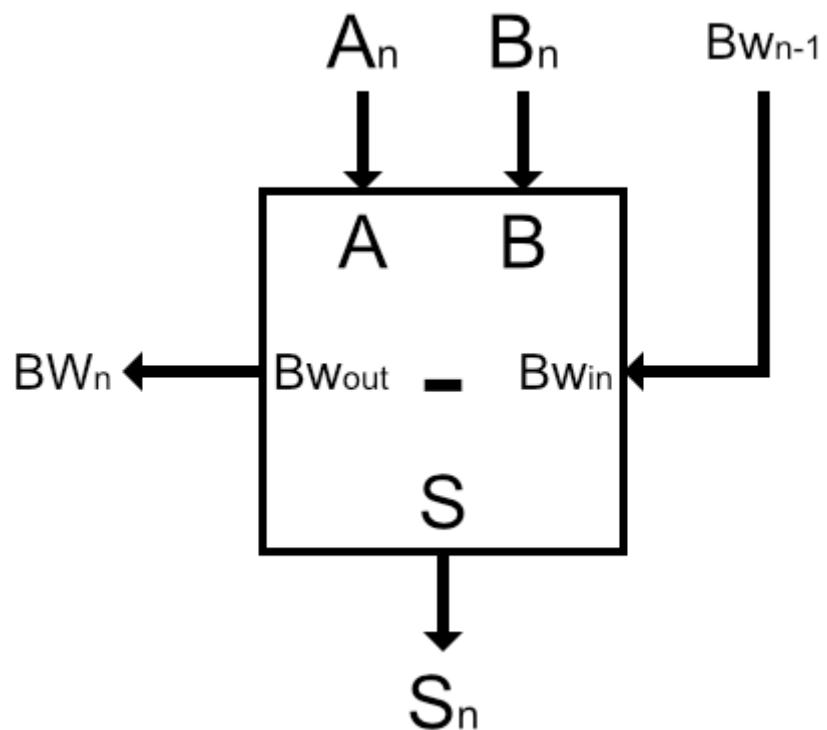


Figura 33 - Módulo Subtrator Completo

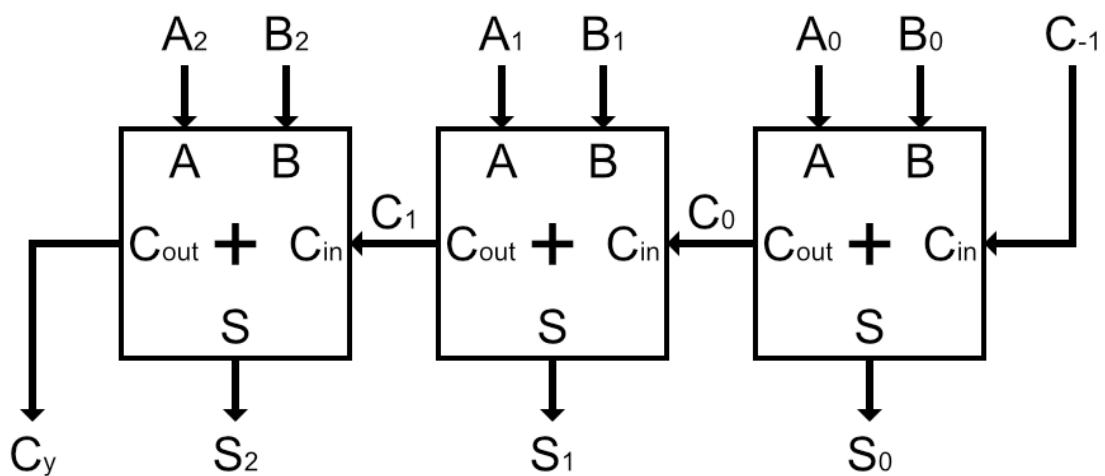


Figura 34 - Somador de Dois Números a Três Bits

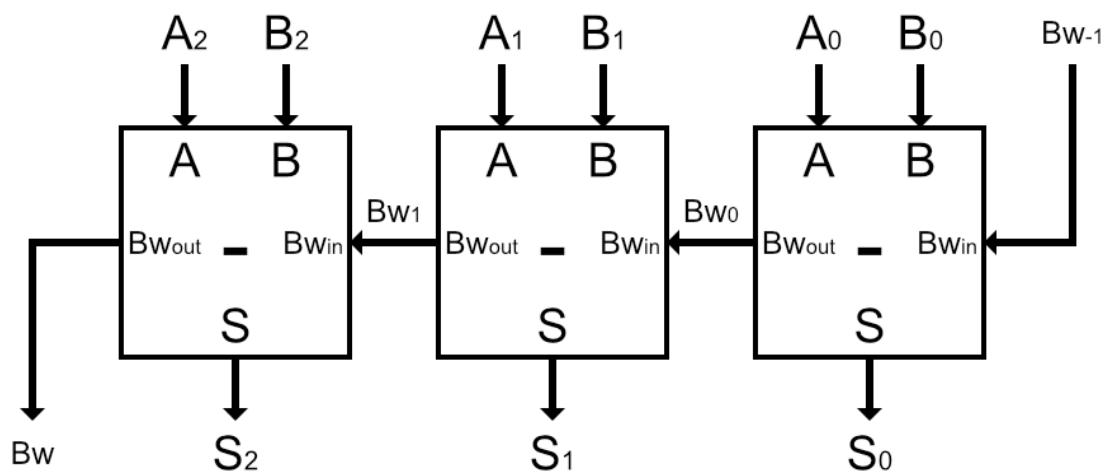


Figura 35 - Subtrator de Dois Números a Três Bits

C_{n-1}	A_n	B_n	C_n	S_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figura 36 - Tabela de Verdade da Soma

B_{n-1}	A_n	B_n	B_n	S_n
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Figura 37 - Tabela de Verdade da Subtração

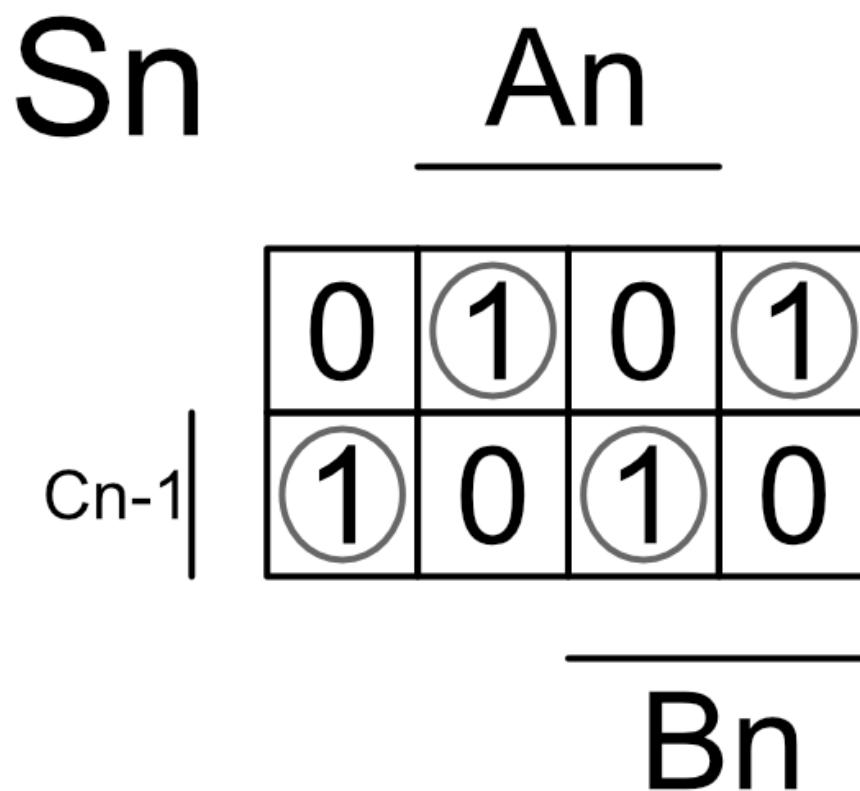


Figura 38 - Mapa de Karnaugh para Sn na Soma

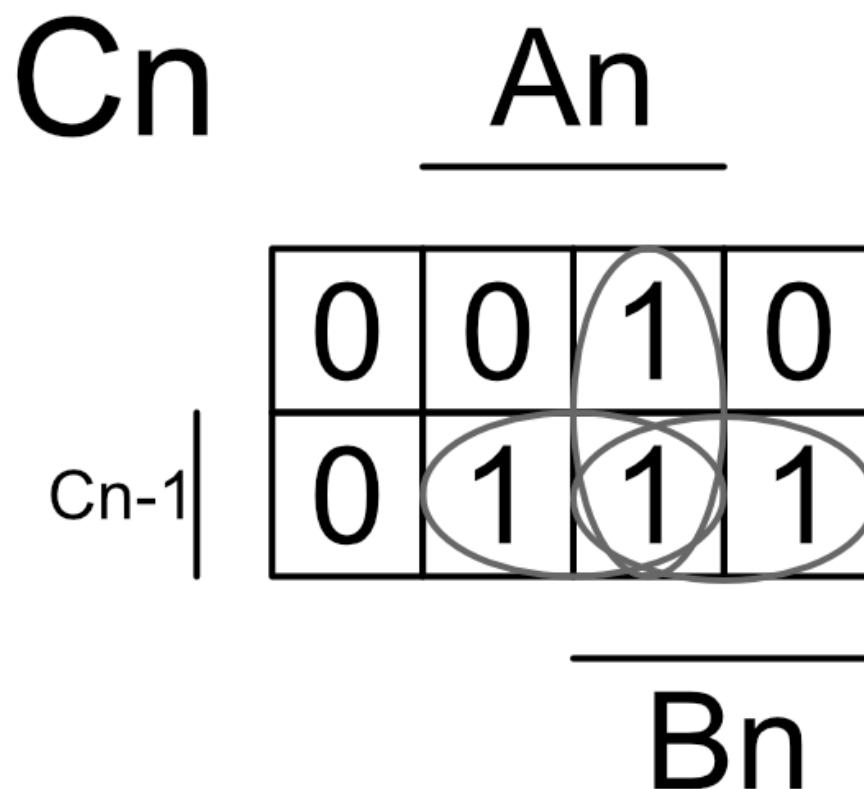


Figura 39 - Mapa de Karnaugh para C_n

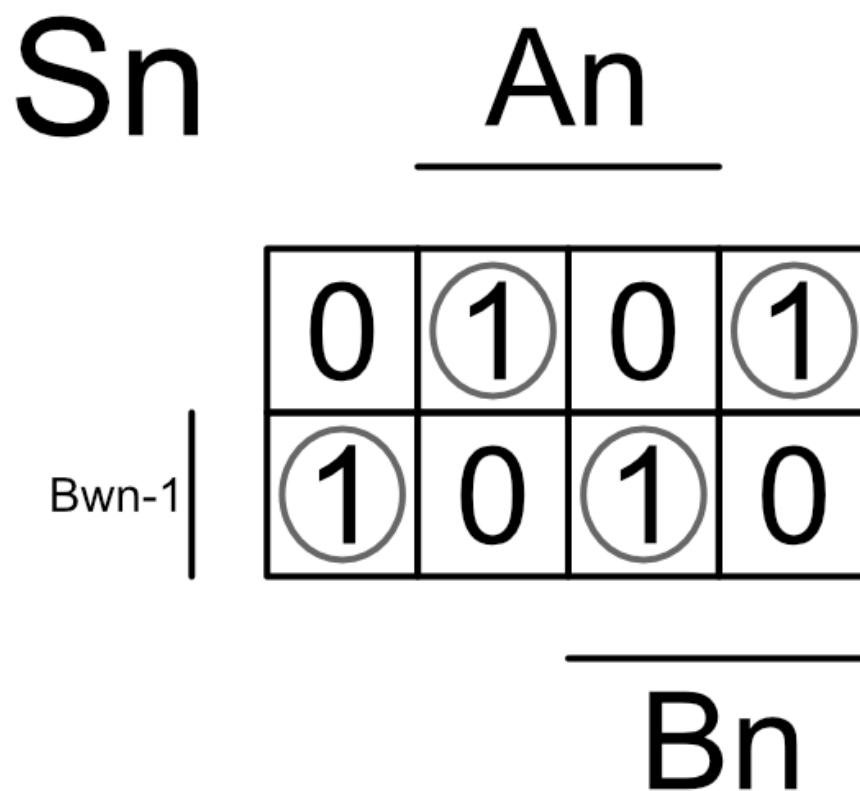


Figura 40 - Mapa de Karnaugh para S_n para Subtração

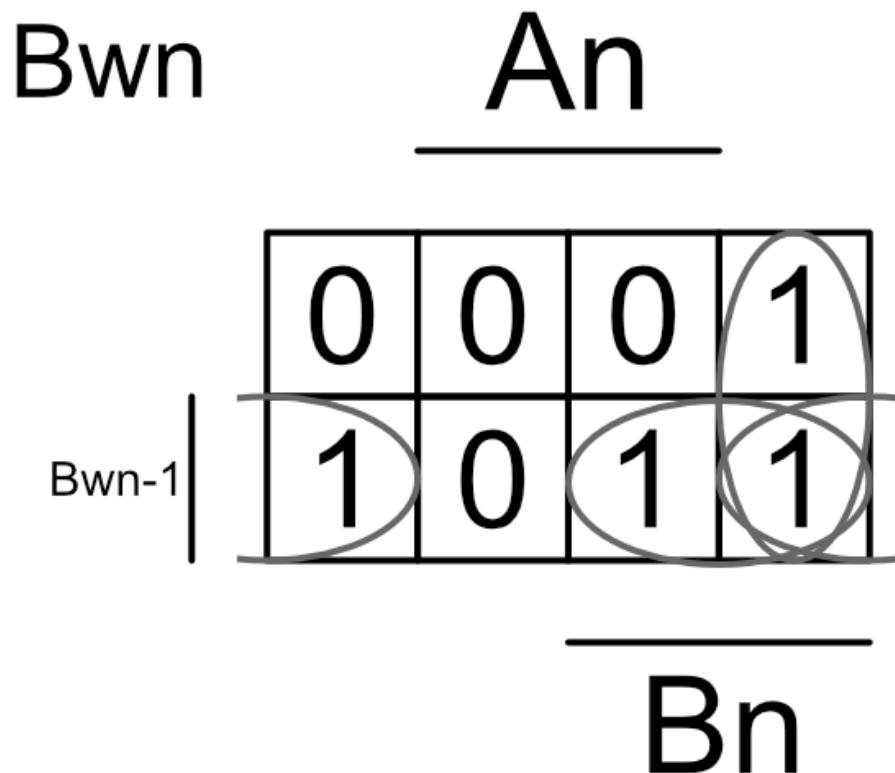


Figura 41 - Mapa de Karnaugh para B_n

$$S_n = \overline{C_{n-1}} \cdot \overline{A_n} \cdot \overline{B_n} + \overline{C_{n-1}} \cdot \overline{A_n} \cdot B_n + C_{n-1} \cdot \overline{A_n} \cdot \overline{B_n} + C_{n-1} \cdot A_n \cdot B_n$$

Figura 42 - Expressão de S_n na Soma

$$C_n = A_n \cdot B_n + C_{n-1} \cdot A_n + C_{n-1} \cdot B_n$$

Figura 43 - Expressão de C_n

$$S_n = \overline{B_{n-1}} \cdot \overline{A_n} \cdot \overline{B_n} + \overline{B_{n-1}} \cdot \overline{A_n} \cdot B_n + B_{n-1} \cdot \overline{A_n} \cdot \overline{B_n} + B_{n-1} \cdot A_n \cdot B_n$$

Figura 44 - Expressão de S_n na Subtração

$$B_{wn} = \overline{A_n} \cdot B_n + B_{wn-1} \cdot \overline{A_n} + B_{wn-1} \cdot B_n$$

Figura 45 - Expressão de B_{wn}

Sel	A_n	F
0	0	0
0	1	1
1	0	1
1	1	0

Figura 46 - Tabela de Verdade do Selector

$$S_n = \overline{C_y B_{wn-1}} \cdot \overline{A_n} \cdot \overline{B_n} + \overline{C_y B_{wn-1}} \cdot \overline{A_n} \cdot B_n + C_y B_{wn-1} \cdot \overline{A_n} \cdot \overline{B_n} + C_y B_{wn-1} \cdot A_n \cdot B_n$$

Figura 47 - Expressão Simplificada do S_n

$$S_n = C_y B_{wn-1} \wedge A_n \wedge B_n$$

Figura 48 - Expressão Final Simplificada do S_n

$$C_y B_w = (Sel \wedge A_n) \cdot B_n + C_y B_{wn-1} \cdot (Sel \wedge A_n) + C_y B_{wn-1} \cdot B_n$$

Figura 49 - Expressão Final do $C_y B_w$

$$O_v = C_y B_{wn-1} \wedge C_y B_{wn-2}$$

Figura 50 - Expressão do O_v

S2	S1	S0	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figura 51 - Tabela de Verdade do Zero

$$\text{Zero} = \overline{S2 \wedge S1 \wedge S0}$$

Figura 52 - Expressão do Zero

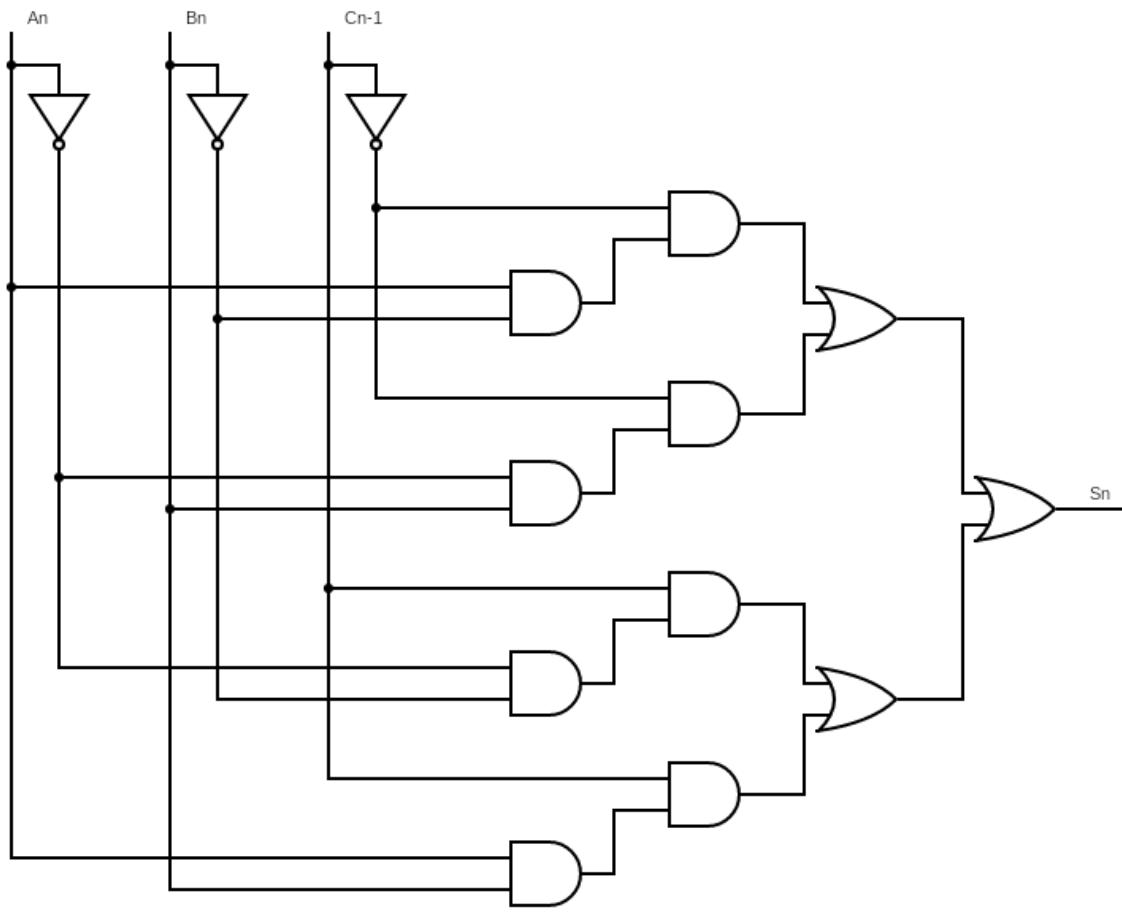


Figura 53 - Circuito Lógico de S_n para a Soma

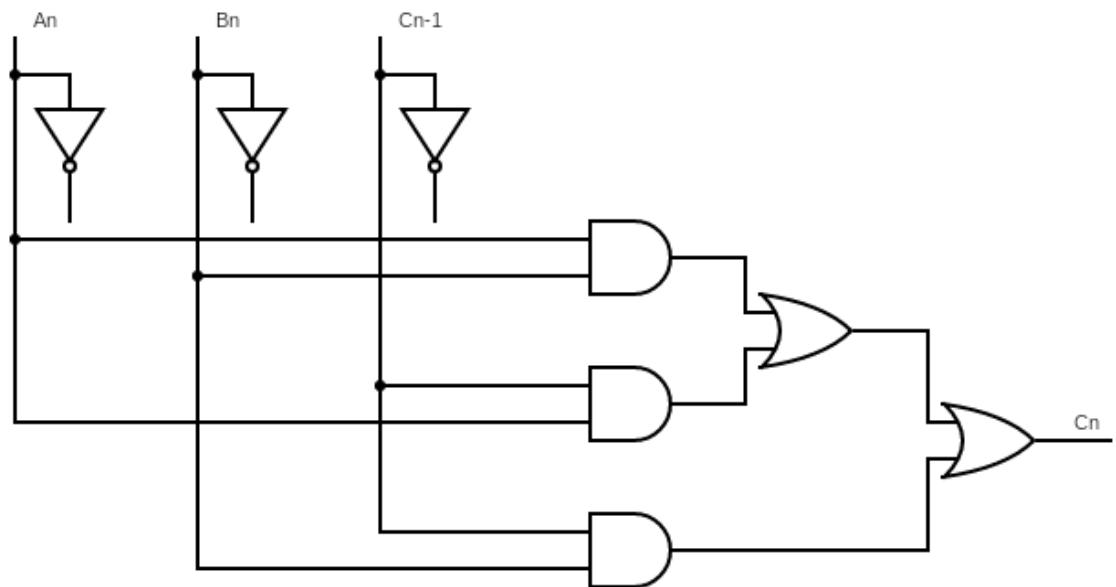


Figura 54 - Circuito Lógico de C_n

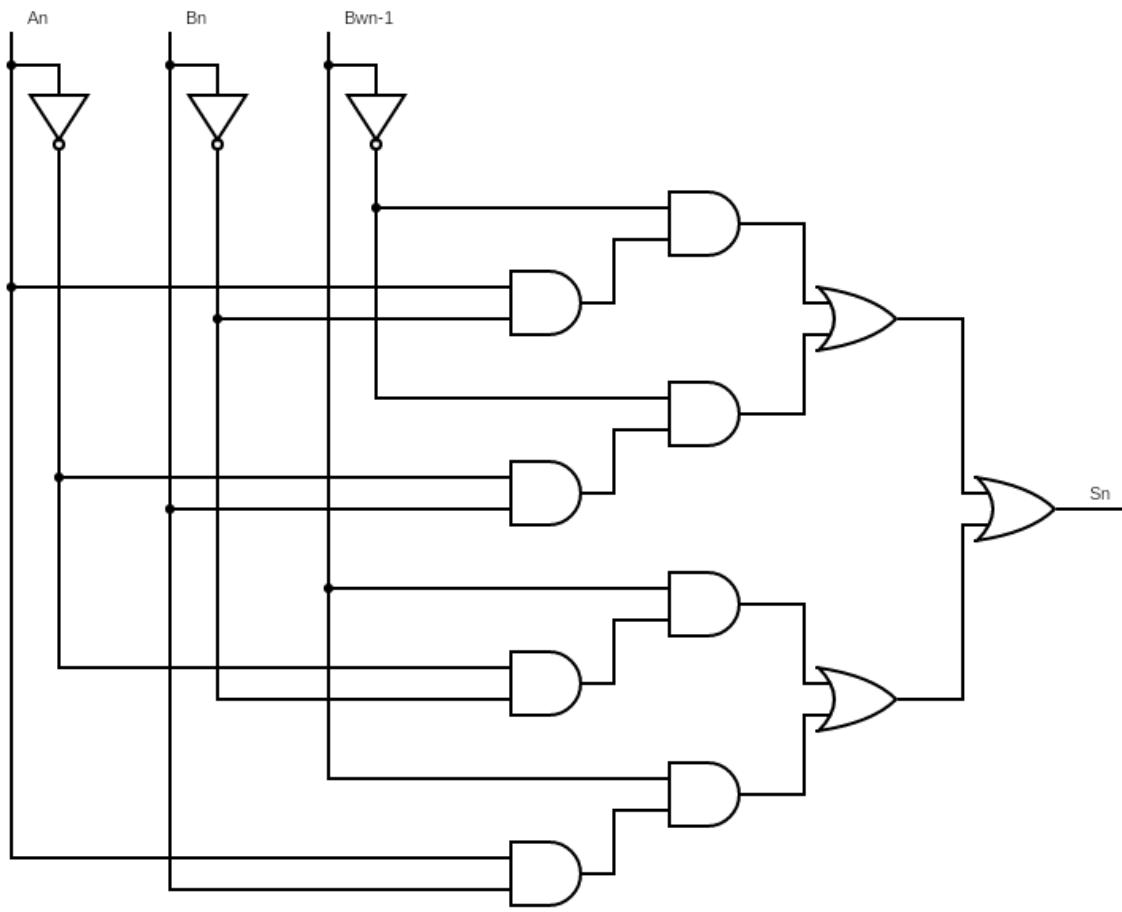


Figura 55 - Circuito Lógico de S_n para a Subtração

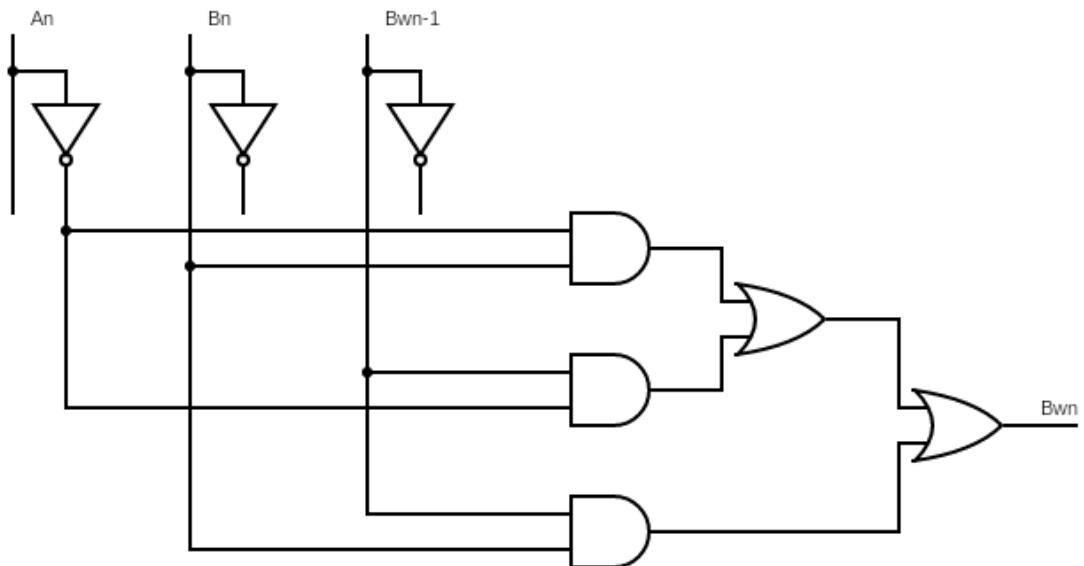


Figura 56 - Circuito Lógico de B_n

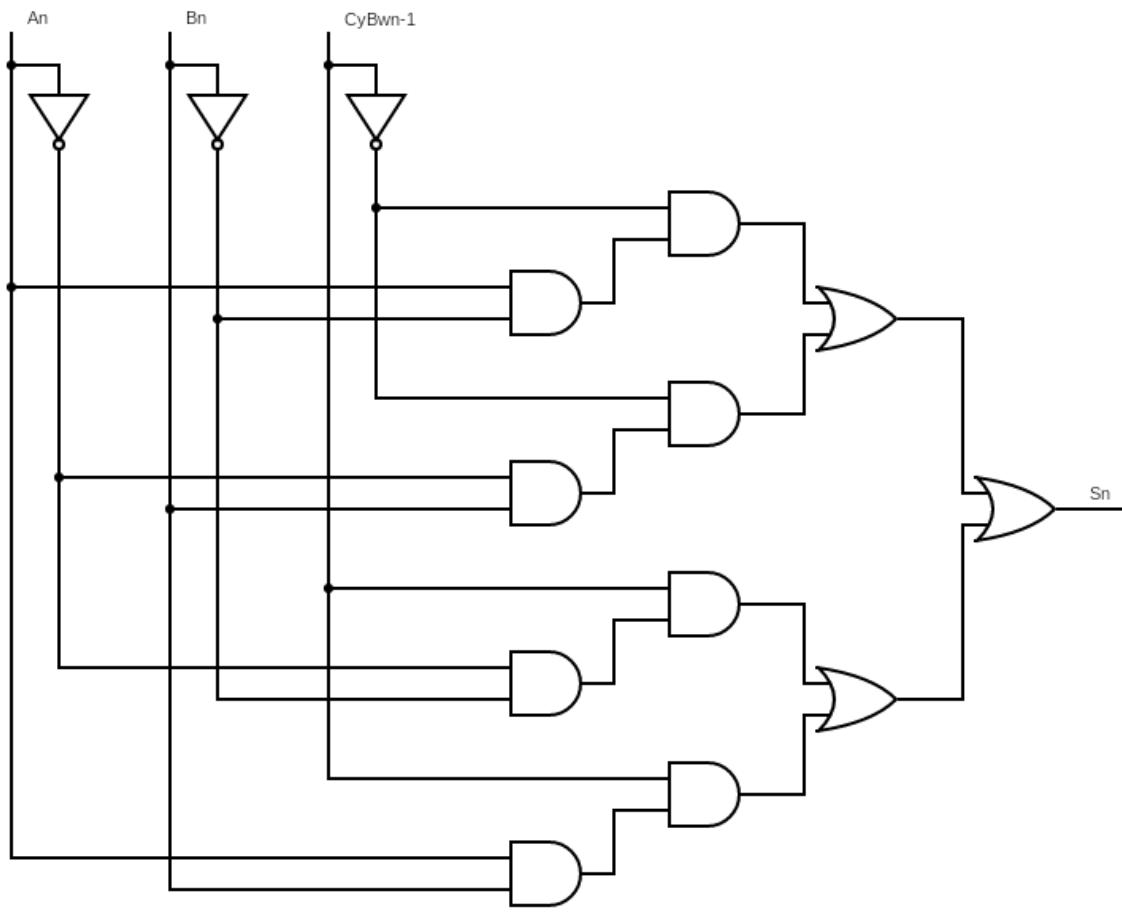


Figura 57 - Circuito Lógico Final de S_n

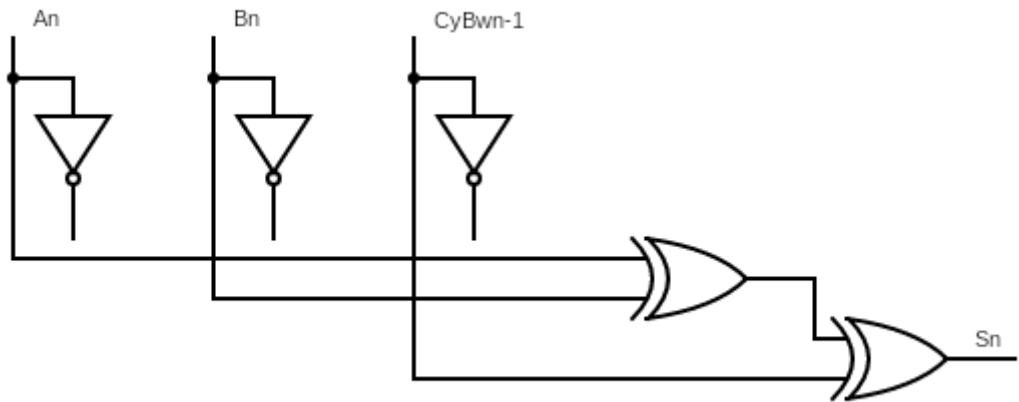


Figura 58 - Circuito Lógico Final Simplificado de S_n

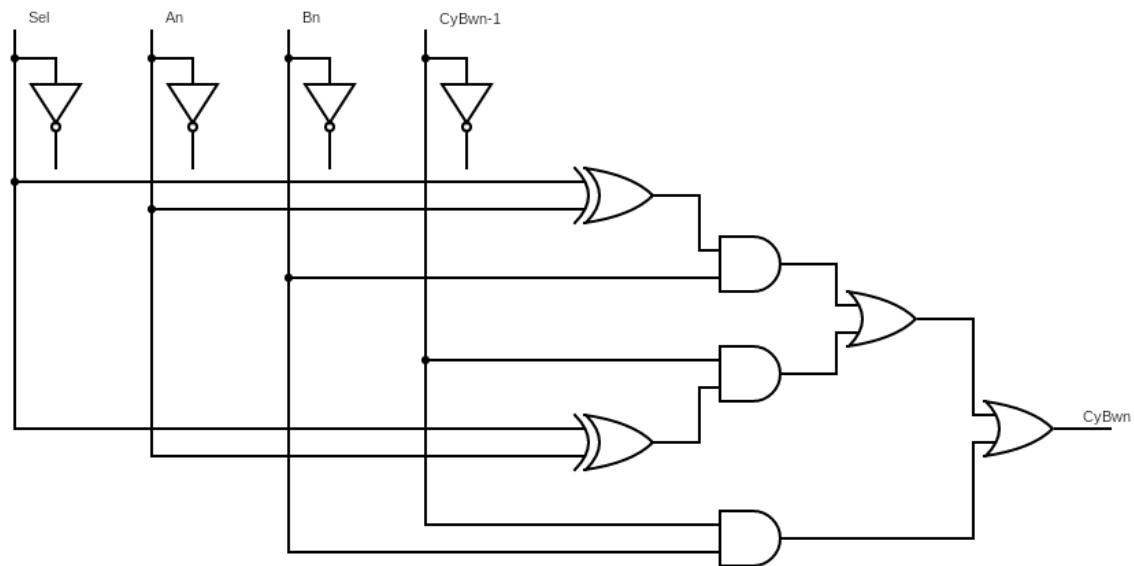


Figura 59 - Circuito Final de CyBw

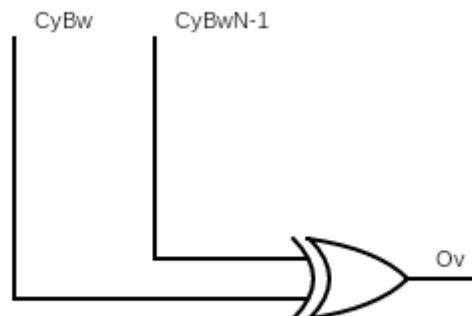


Figura 60 - Circuito Lógico de Ov

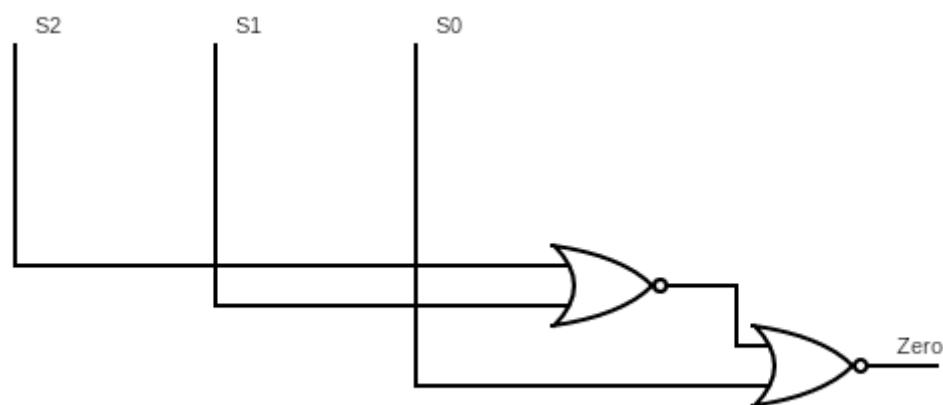


Figura 61 - Circuito Lógico de Zero

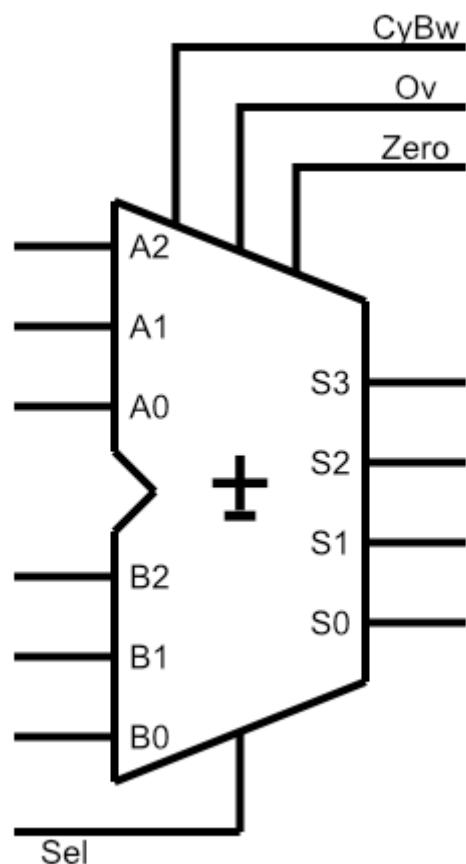


Figura 62 - Símbolo Lógico Somador e Subtrator

Código

Começamos por definir os pinos para cada um dos bits de cada uma das variáveis de entrada, Ao, A1 e A2, Bo, B1, B2. Também os pinos para os bits das variáveis de saída, So, S1 e S2. Por fim, definimos os pinos das *flags*, CarryBorrow, Overflow e Zero. (Ver Figura 63)

```
#define pinSelector A0

#define pinA0 2
#define pinA1 3
#define pinA2 4
#define pinB0 5
#define pinB1 6
#define pinB2 7
#define pinS0 8
#define pinS1 9
#define pinS2 10

#define pinL_CarryBorrow 11
#define pinL_Overflow 12
#define pinL_Zero 13
```

Figura 63 - Definição de Pinos

Definimos as variáveis a serem usadas, neste caso dado que só podem tomar valores de 0 ou 1, estas serão booleanas. (Ver Figura 64)

```
boolean Sel, A_0, A_1, A_2, B_0, B_1, B_2, S_0, S_1, S_2, CBw_0, CBw_1, L_CyBw, L_Ov, L_Zero;
```

Figura 64 - Definição de Variáveis

É implementado o módulo que afecta cada bit de S, transformado em variável booleana, ou seja, So, S1 e S2. Dado que cada cada variável Sn também depende do valor prévio de *carry* ou *borrow*, dependendo da operação em questão, e produz um novo *carry* ou *borrow*, é necessário afectar ambos mas a expressão ser modular.

Dado isto, os argumentos são as variáveis de entrada, gerais, An e Bn, e o CyBwAnterior. Temos de passar as variáveis que são afectadas por referência, ou seja, Sn e CyBw com um asterisco, *, que indica esta referência.

No corpo do método, temos as expressões arranjadas previamente a afectar estas referências também com o asterisco. (Ver Figura 65)

```
void calculateResult(boolean Sel, boolean An, boolean Bn, boolean CnBwnAnterior, boolean *Sn, boolean *CyBw)
{
    *Sn = CnBwnAnterior ^ An ^ Bn;
    *CyBw = (Sel ^ An) & Bn | CnBwnAnterior & (Sel ^ An) | CnBwnAnterior & Bn;
}
```

Figura 65 - Implementação do calculateResult

Definimos um método que afeta as variáveis que simbolizam as *flags*, implementando as expressões arranjadas previamente. (*Ver Figura 66 e Figura 67*)

```
boolean flagOverflow()
{
    return CBw_1 ^ L_CyBw;
}
```

Figura 66 - Implementação de flagOverflow

```
boolean flagZero()
{
    return !(S_0 | S_1 | S_2);
}
```

Figura 67 - Implementação de flagZero

As variáveis que representam estas flags são afetadas numa só chamada por um método void que, por sua vez, chama os métodos implementados previamente que retornam os valores das expressões. (*Ver Figura 68*)

```
void setFlags()
{
    L_Ov = flagOverflow();
    L_Zero = flagZero();
}
```

Figura 68 - Implementação de setFlags

Também implementamos uma função para ler os inputs, ou seja, para obter os valores correntes de cada bit das variáveis de entrada. (*Ver Figura 69*)

```
void readInputs()
{
    Sel = digitalRead(pinSelector);
    A_0 = digitalRead(pinA0);
    A_1 = digitalRead(pinA1);
    A_2 = digitalRead(pinA2);
    B_0 = digitalRead(pinB0);
    B_1 = digitalRead(pinB1);
    B_2 = digitalRead(pinB2);
}
```

Figura 69 - Implementação de readInputs

Implementamos uma função para escrever para as variáveis de saída. (Ver Figura 70)

```
void writeOutputs()
{
    digitalWrite(pinS0, S_0);
    digitalWrite(pinS1, S_1);
    digitalWrite(pinS2, S_2);
    digitalWrite(pinL_CarryBorrow, L_CyBw);
    digitalWrite(pinL_Overflow, L_Ov);
    digitalWrite(pinL_Zero, L_Zero);
}
```

Figura 70 - Implementação de writeOutputs

No *setup* temos de ler definir o modo dos pinos previamente definidos (Ver Figura 71)

```
void setup() {
    pinMode(pinSelector, INPUT);
    pinMode(pinA0, INPUT);
    pinMode(pinA1, INPUT);
    pinMode(pinA2, INPUT);
    pinMode(pinB0, INPUT);
    pinMode(pinB1, INPUT);
    pinMode(pinB2, INPUT);

    pinMode(pinS0, OUTPUT);
    pinMode(pinS1, OUTPUT);
    pinMode(pinS2, OUTPUT);
    pinMode(pinL_CarryBorrow, OUTPUT);
    pinMode(pinL_Overflow, OUTPUT);
    pinMode(pinL_Zero, OUTPUT);
}
```

Figura 71 - Setup

Por fim, temos o loop do programa. Neste, temos apenas de ler os valores de entrada, chamar a função que calcula os resultados para cada variável de S. Passamos as respectivas variáveis de entrada, An e Bn, os CarryBorrow's anteriores, assim como as referências para o Sn apropriado e a referência para o CarryBorrow seguinte. Finalmente, afetamos as flags e enviamos os resultados para os pinos. (Ver Figura 72)

```
void loop() {
    readInputs();

    calculateResult(Sel, A_0, B_0, 0, &S_0, &CBw_0);
    calculateResult(Sel, A_1, B_1, CBw_0, &S_1, &CBw_1);
    calculateResult(Sel, A_2, B_2, CBw_1, &S_2, &L_CyBw);

    setFlags();
    writeOutputs();
}
```

Figura 72 - Loop

Montagem

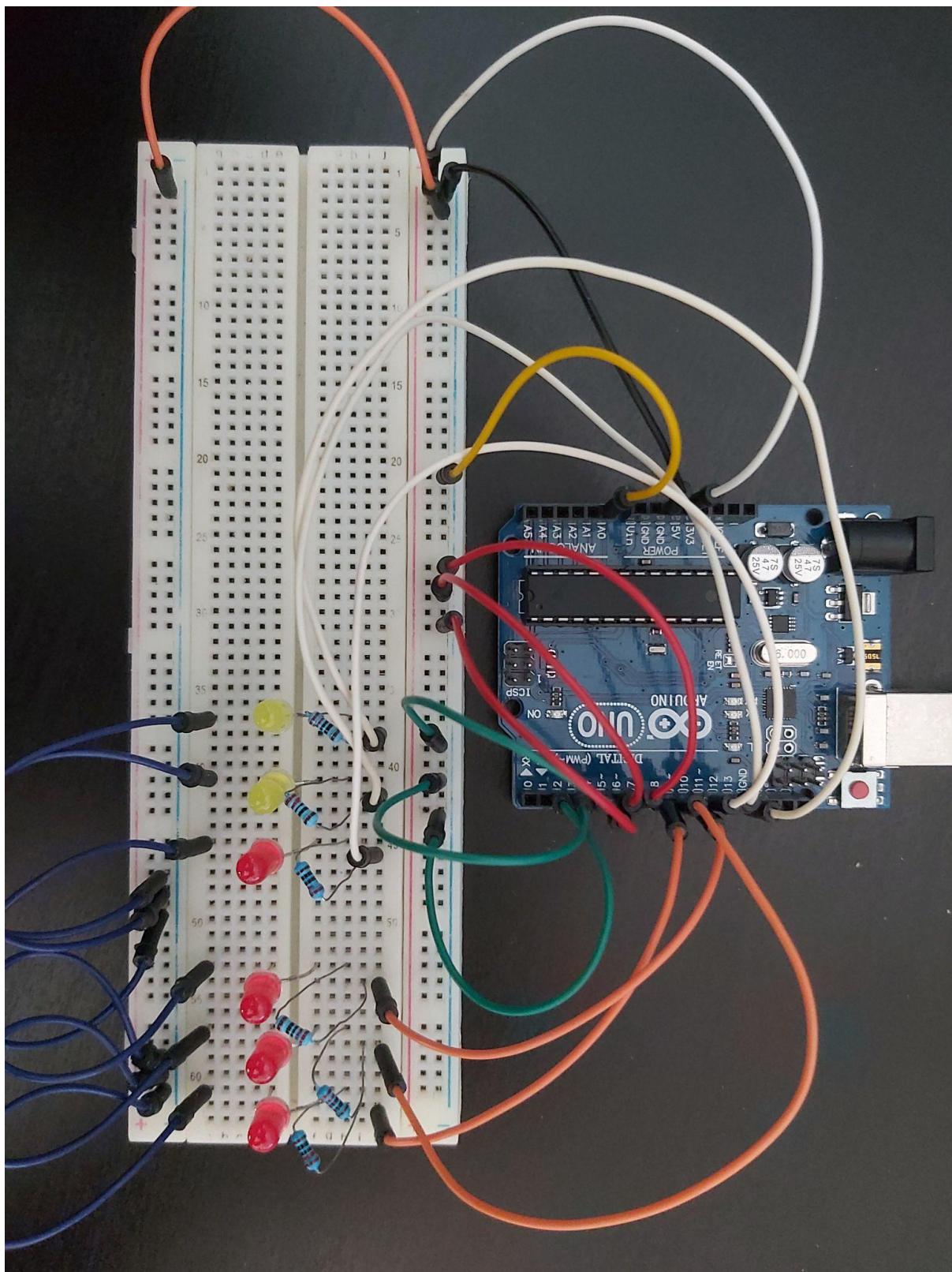


Figura 73 - Montagem Geral

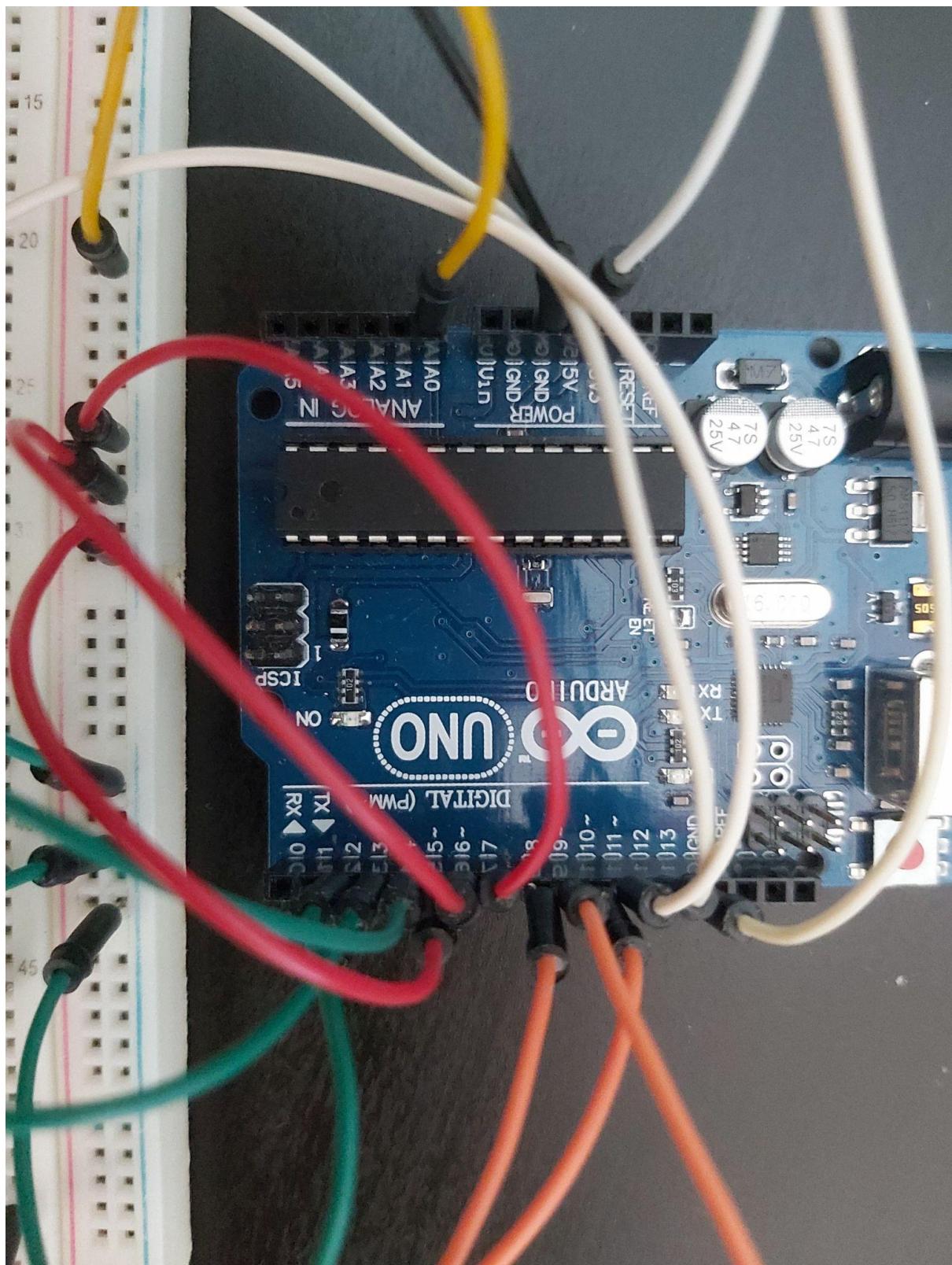


Figura 74 - Montagem Ardúino 1

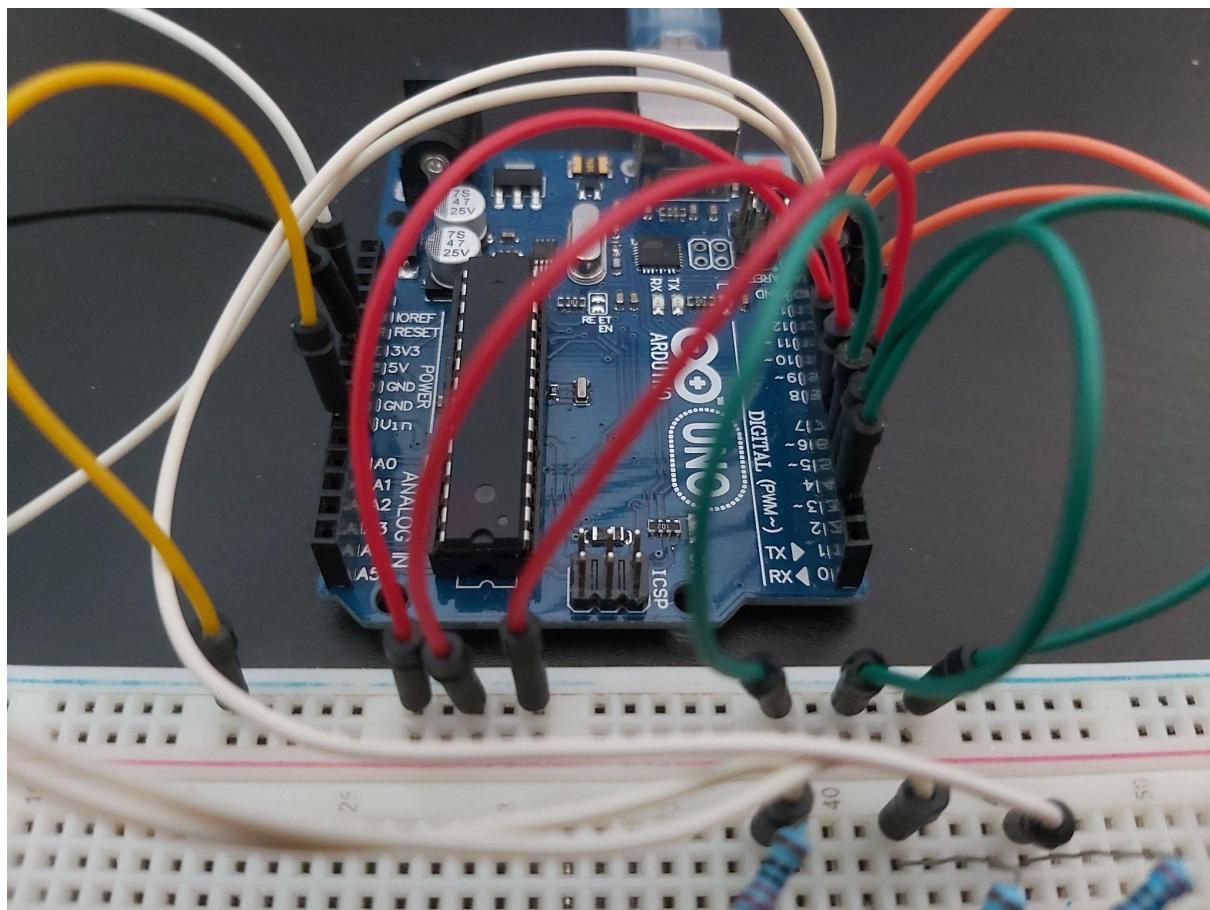


Figura 75 - Montagem Arduíno 2

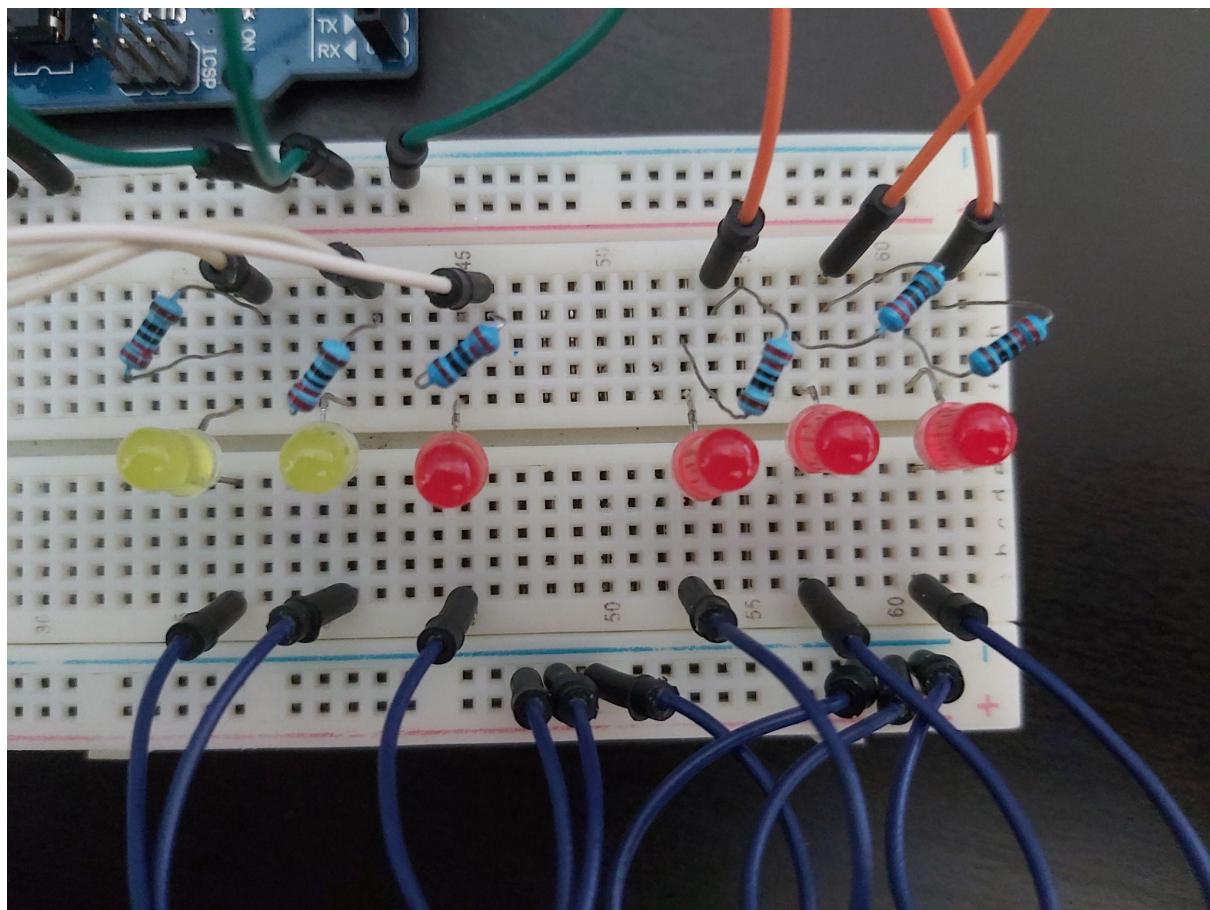


Figura 76 - Montagem Saídas 1

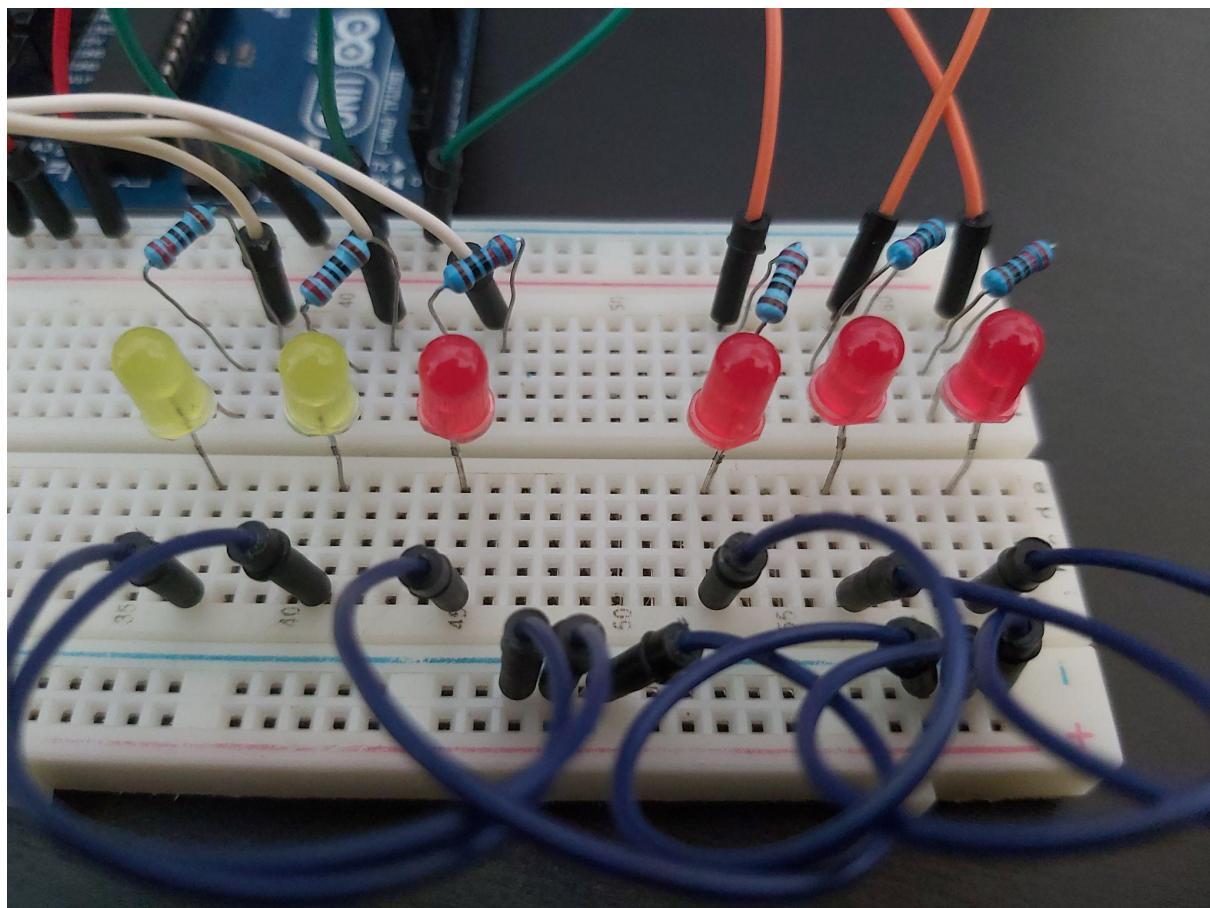


Figura 77 - Montagem Saídas 2

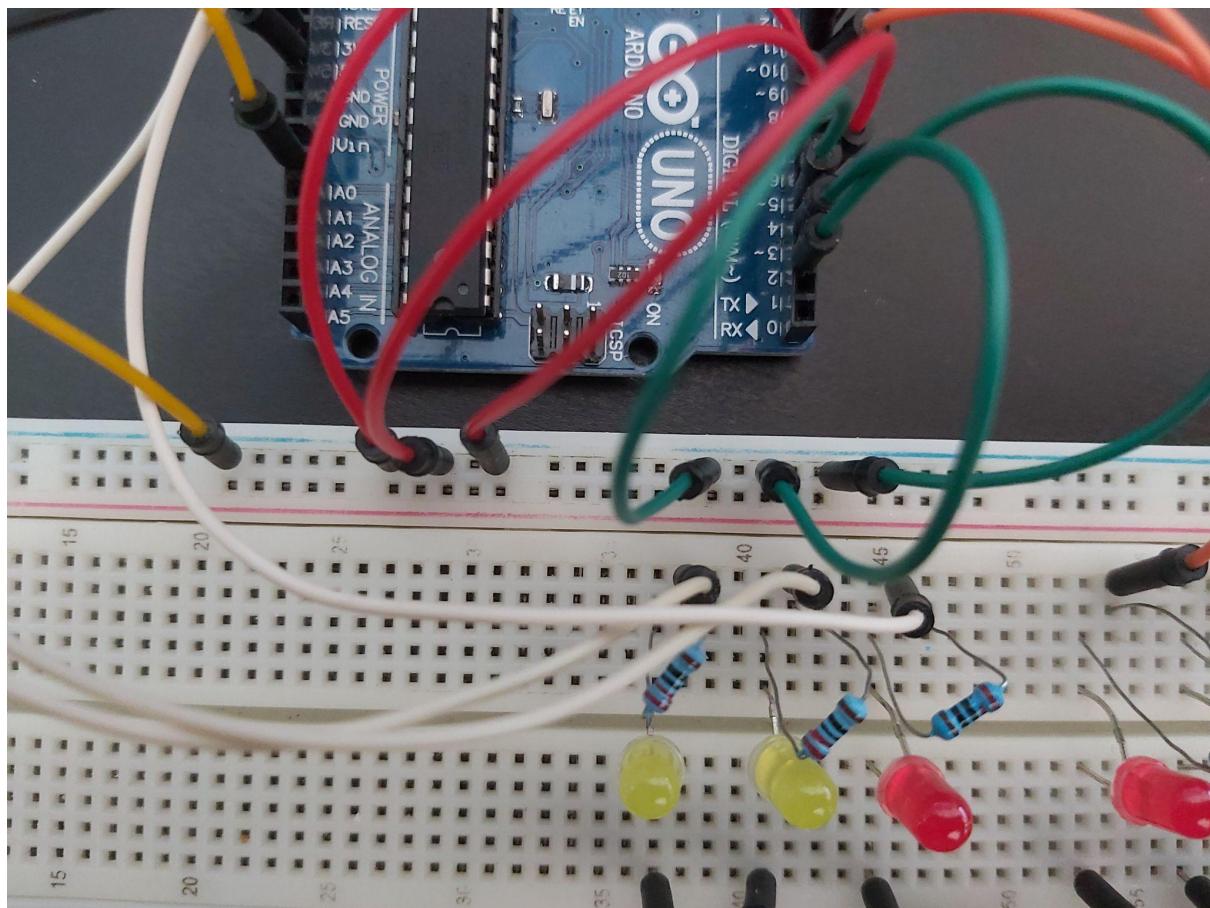


Figura 78 - Montagem Entradas 1

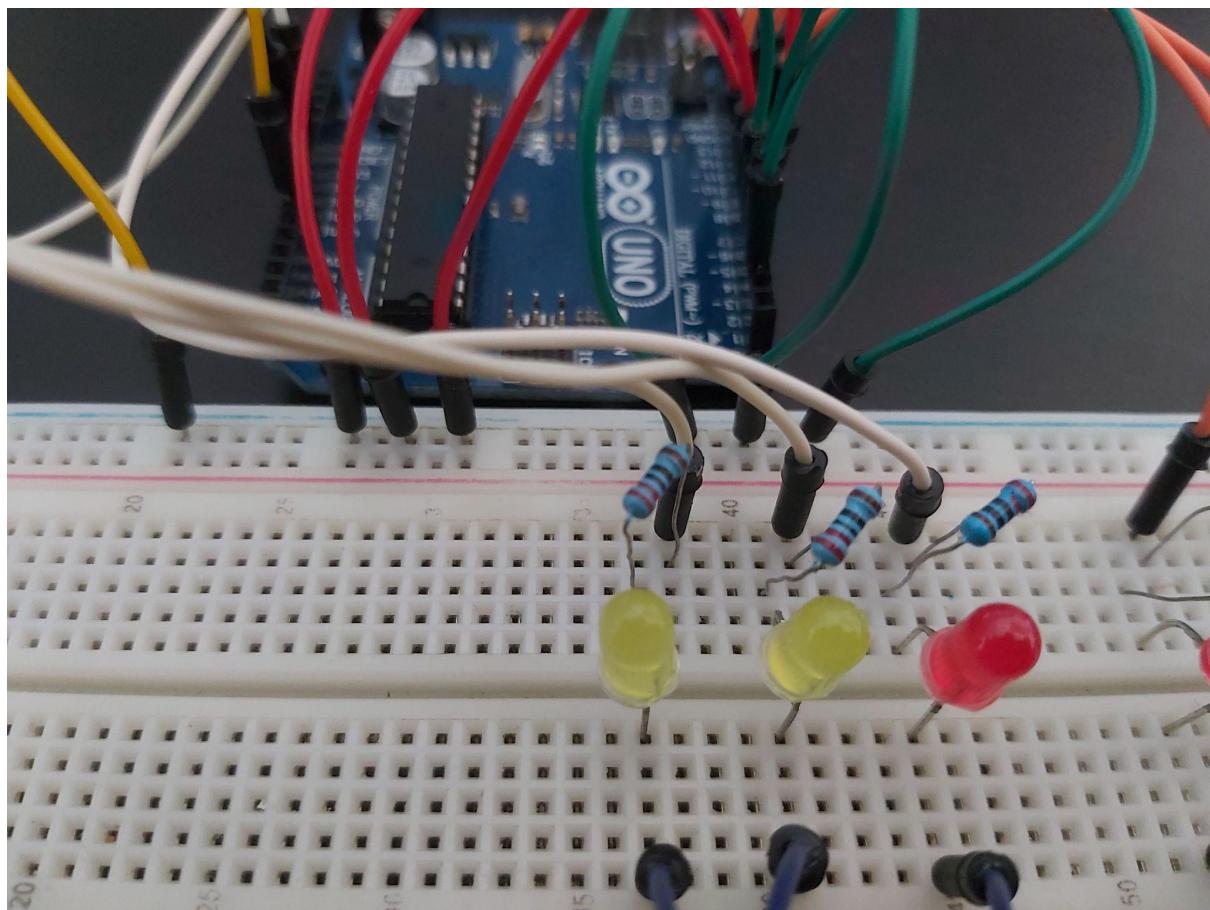


Figura 79 - Montagem Entradas 2

Testes

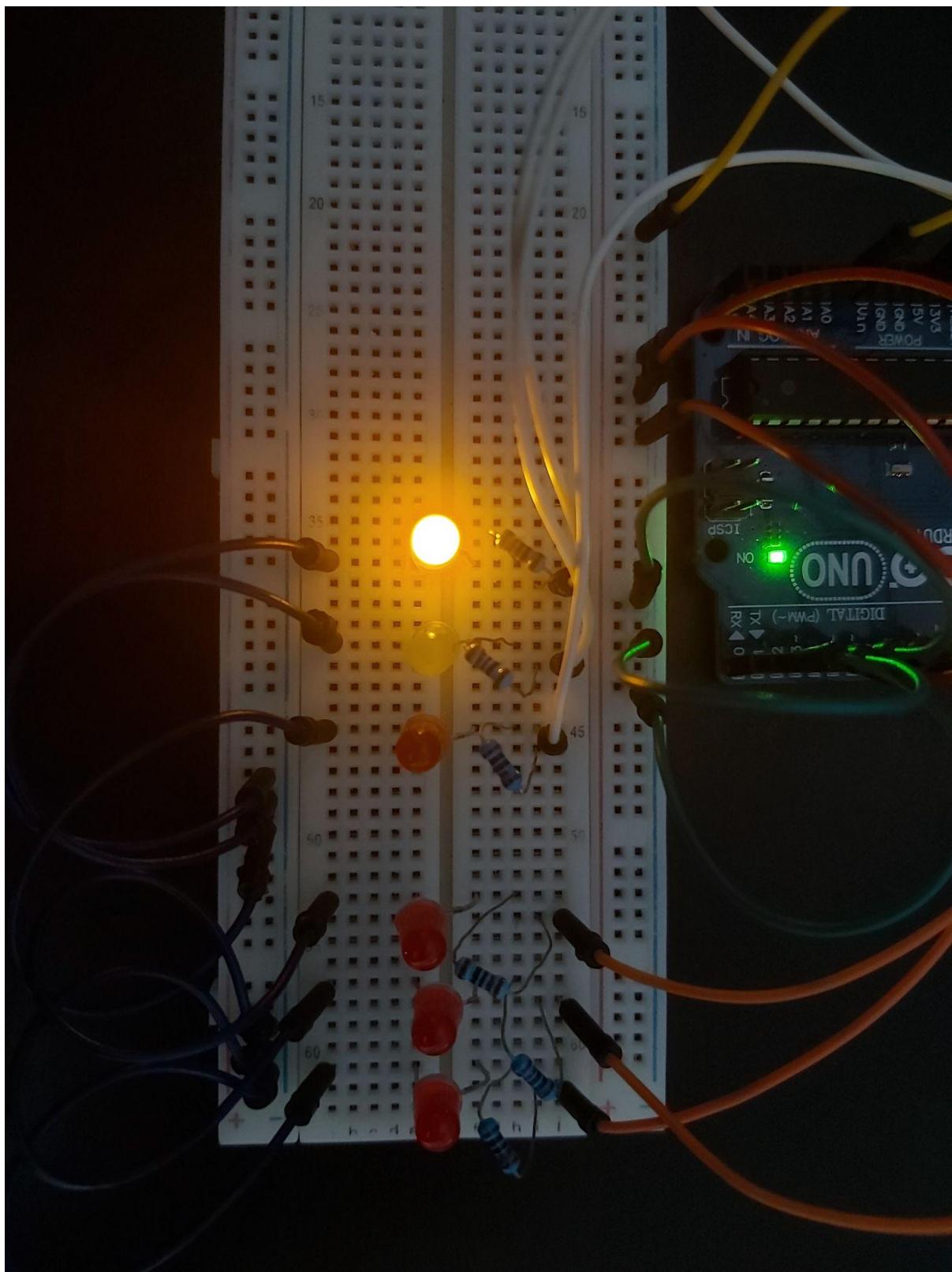


Figura 8o - Testes Flag Zero

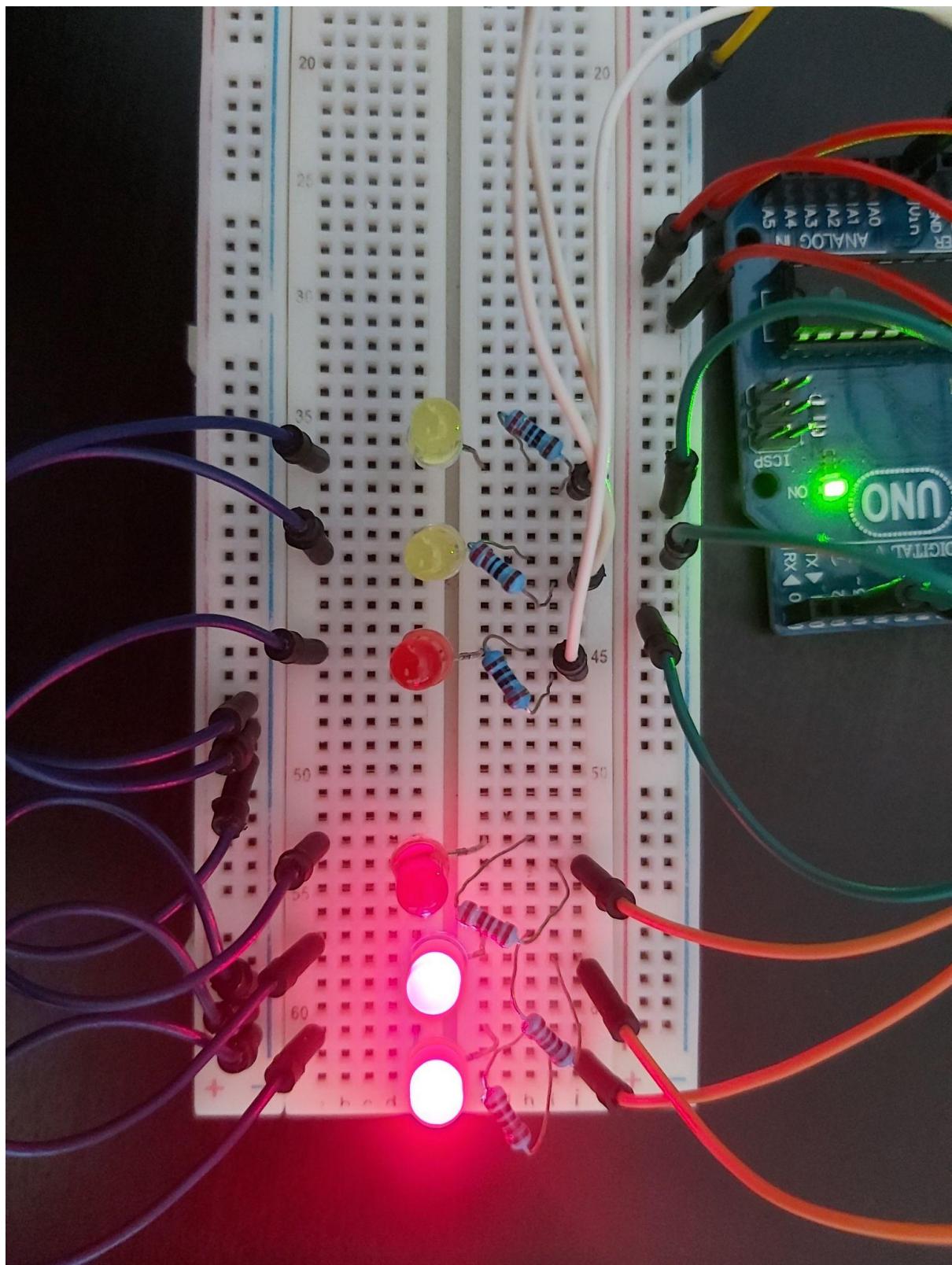


Figura 81 - Testes Soma: A = 001, B = 010, S = 011

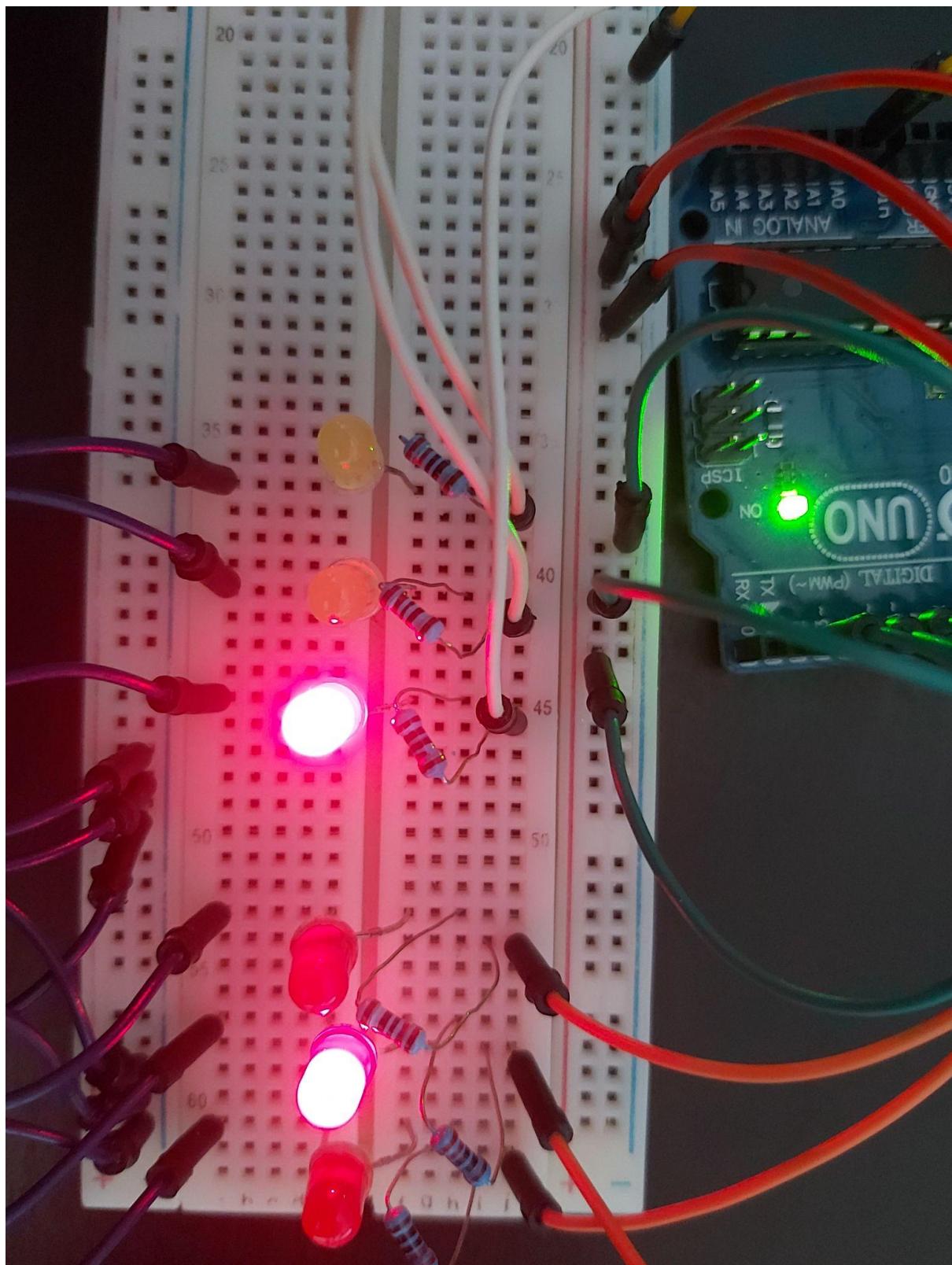


Figura 82 - Testes Soma: A = 011, B = 111, S = 010, Cy = 1

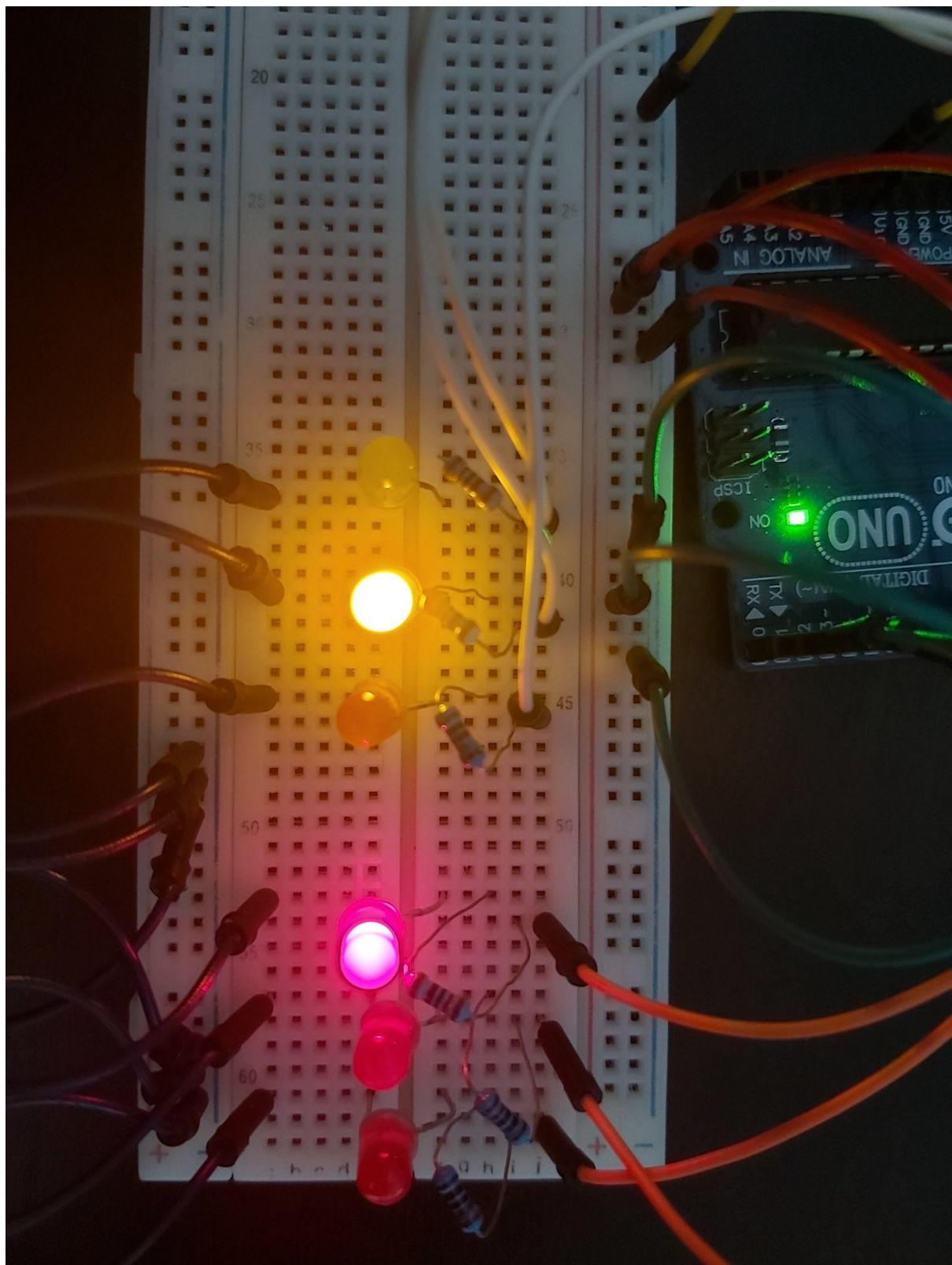


Figura 83 - Testes Soma: A = 010, B = 010, S = 100, Ov = 1

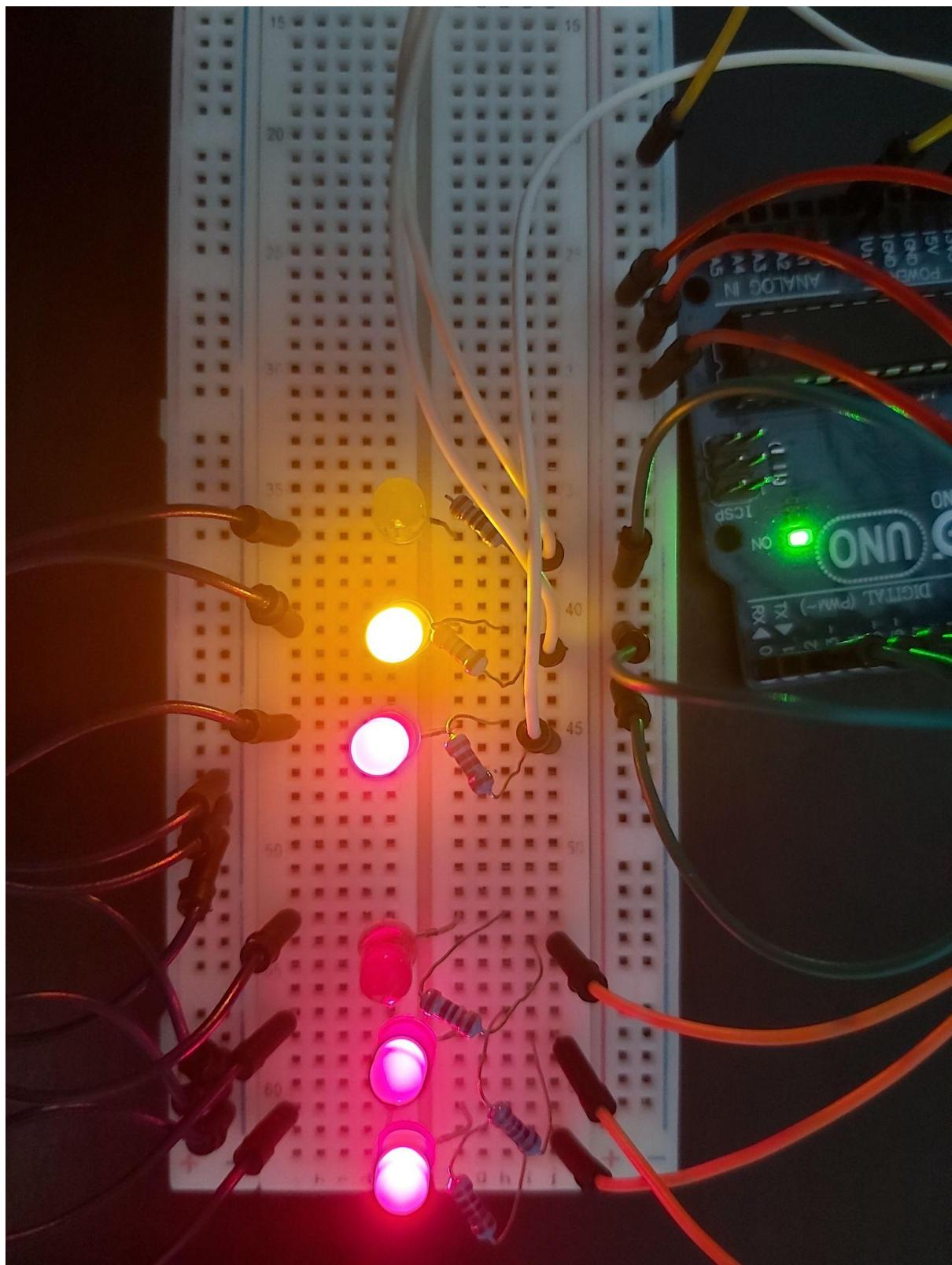


Figura 84 - Testes Soma: A = 111, B = 100, S = 011, Cy = 1, Ov = 1

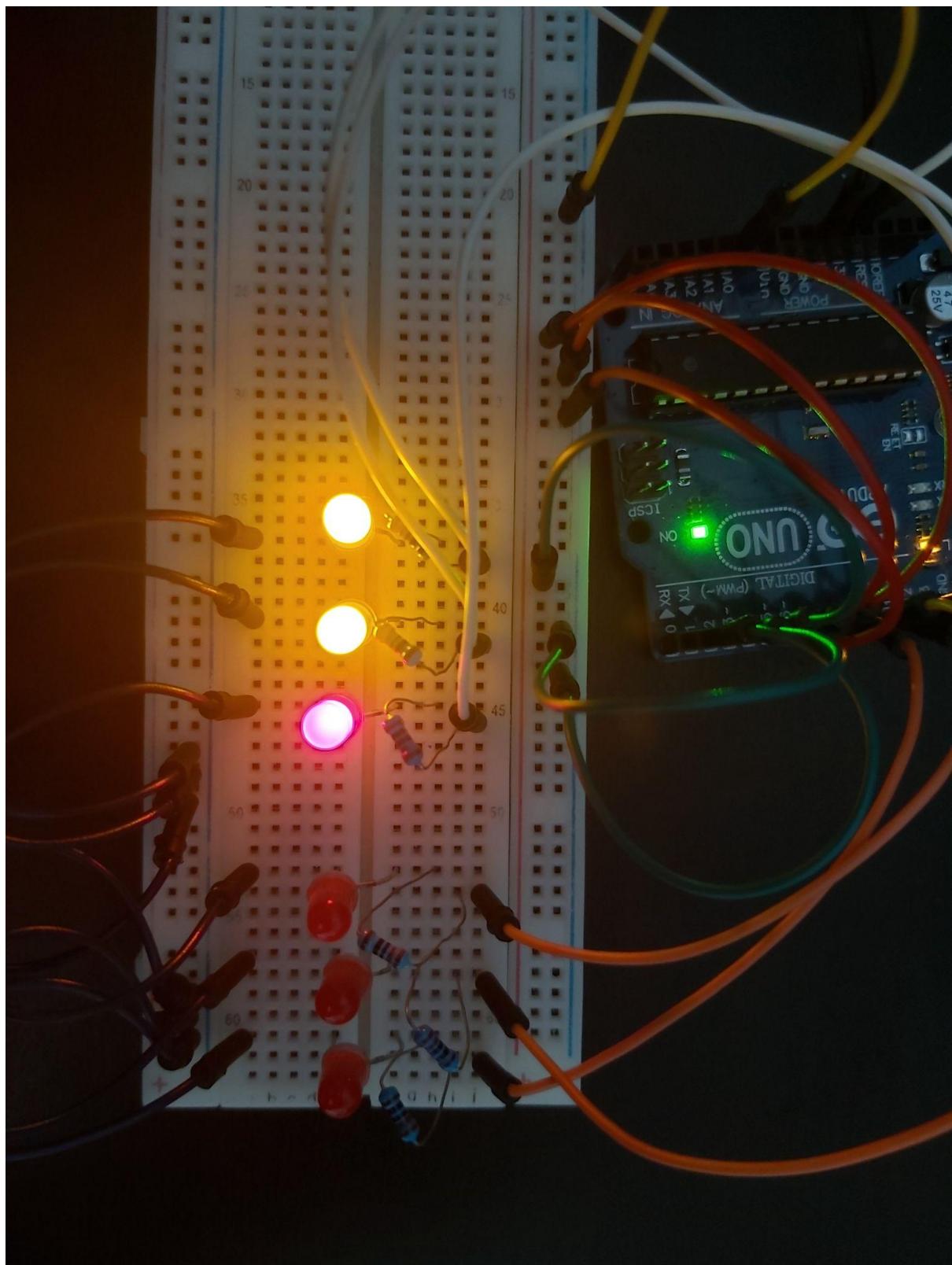


Figura 85 - Testes Soma: A = 100, B = 100, S = 000, Cy = 1, Bw = 1, Zero = 1

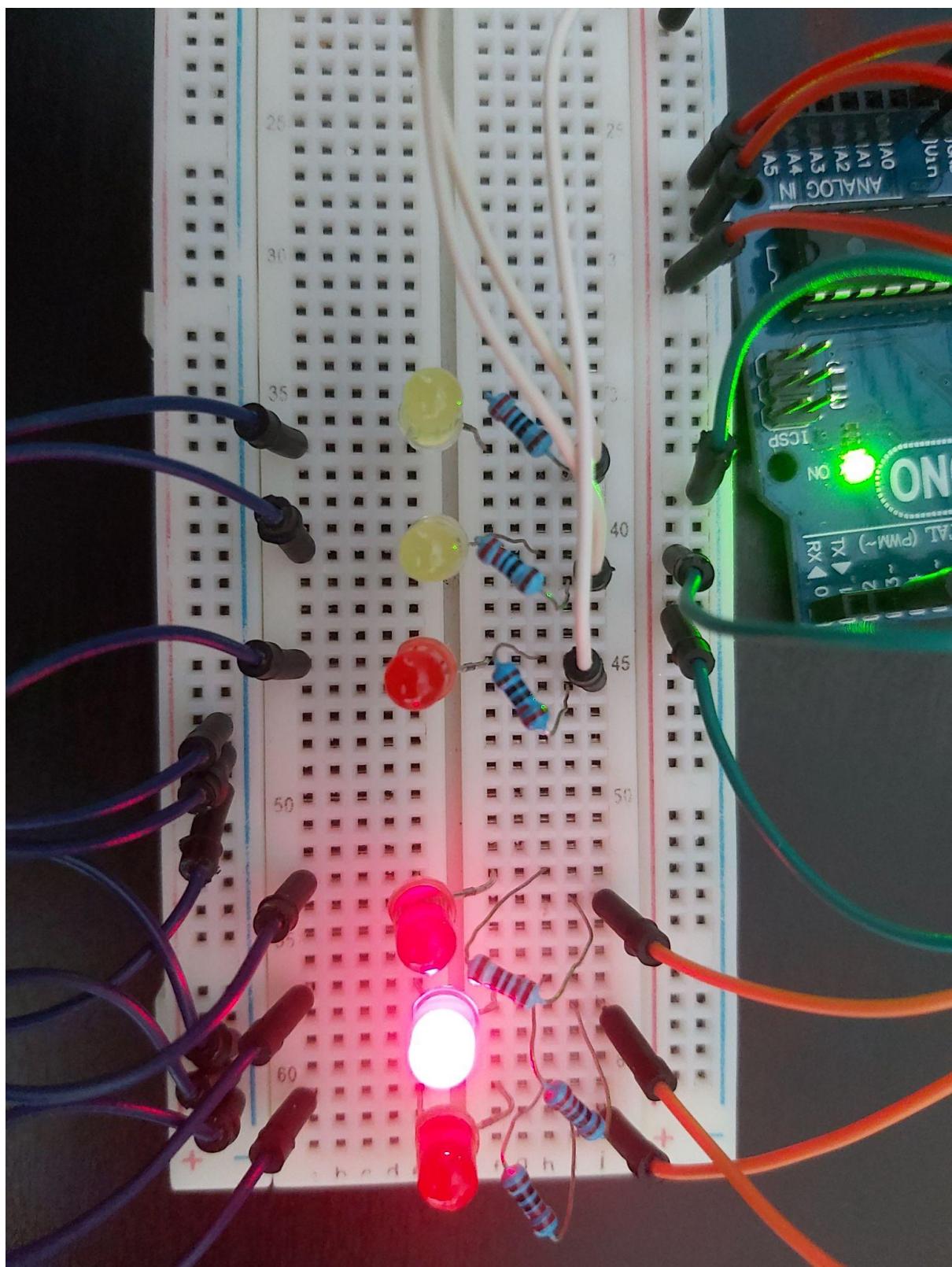


Figura 86 - Testes Subtração: A = 011, B = 001, S = 010

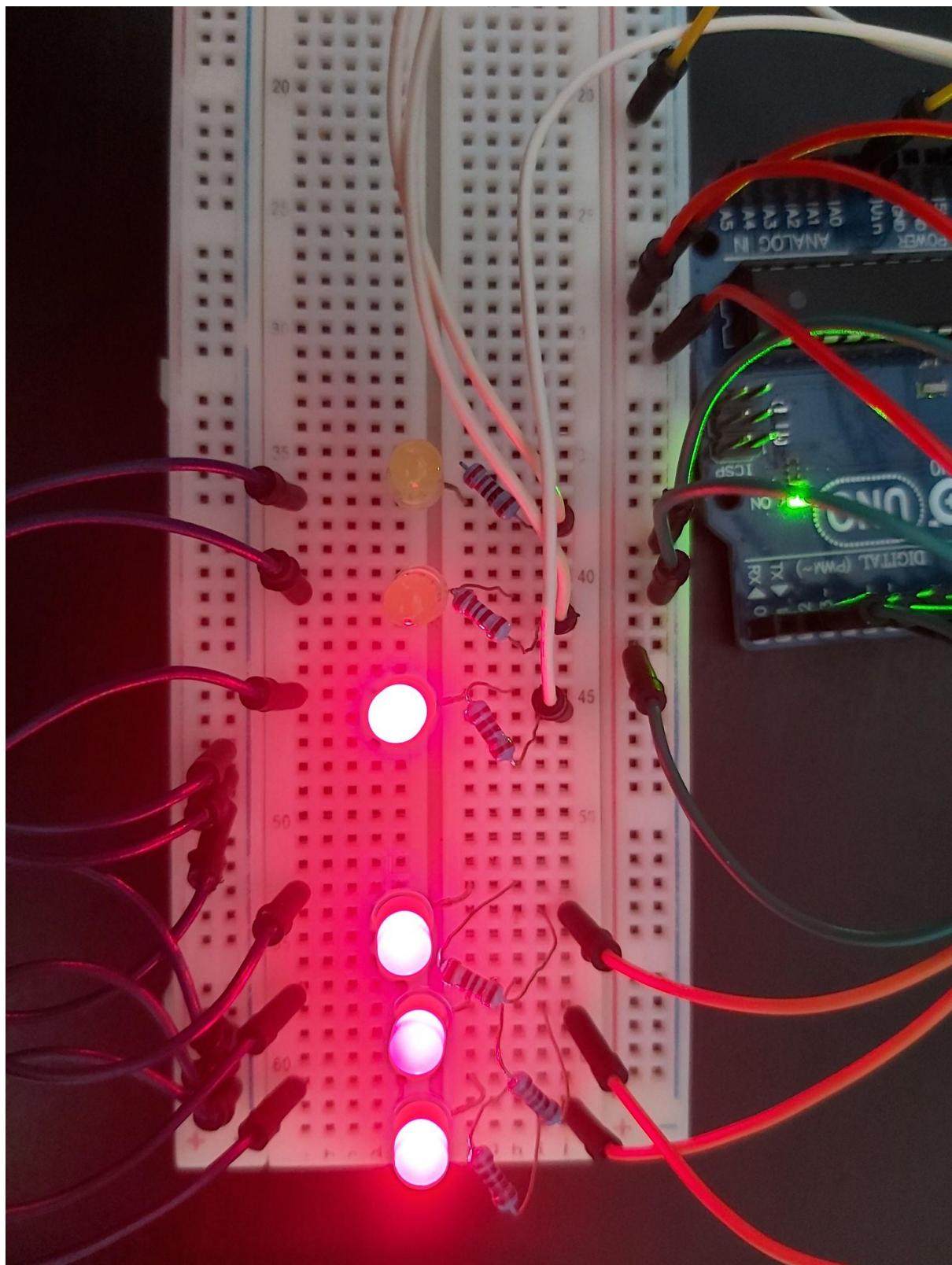


Figura 87 - Testes Subtração: A = 001, B = 010, S = 111, Bw = 1

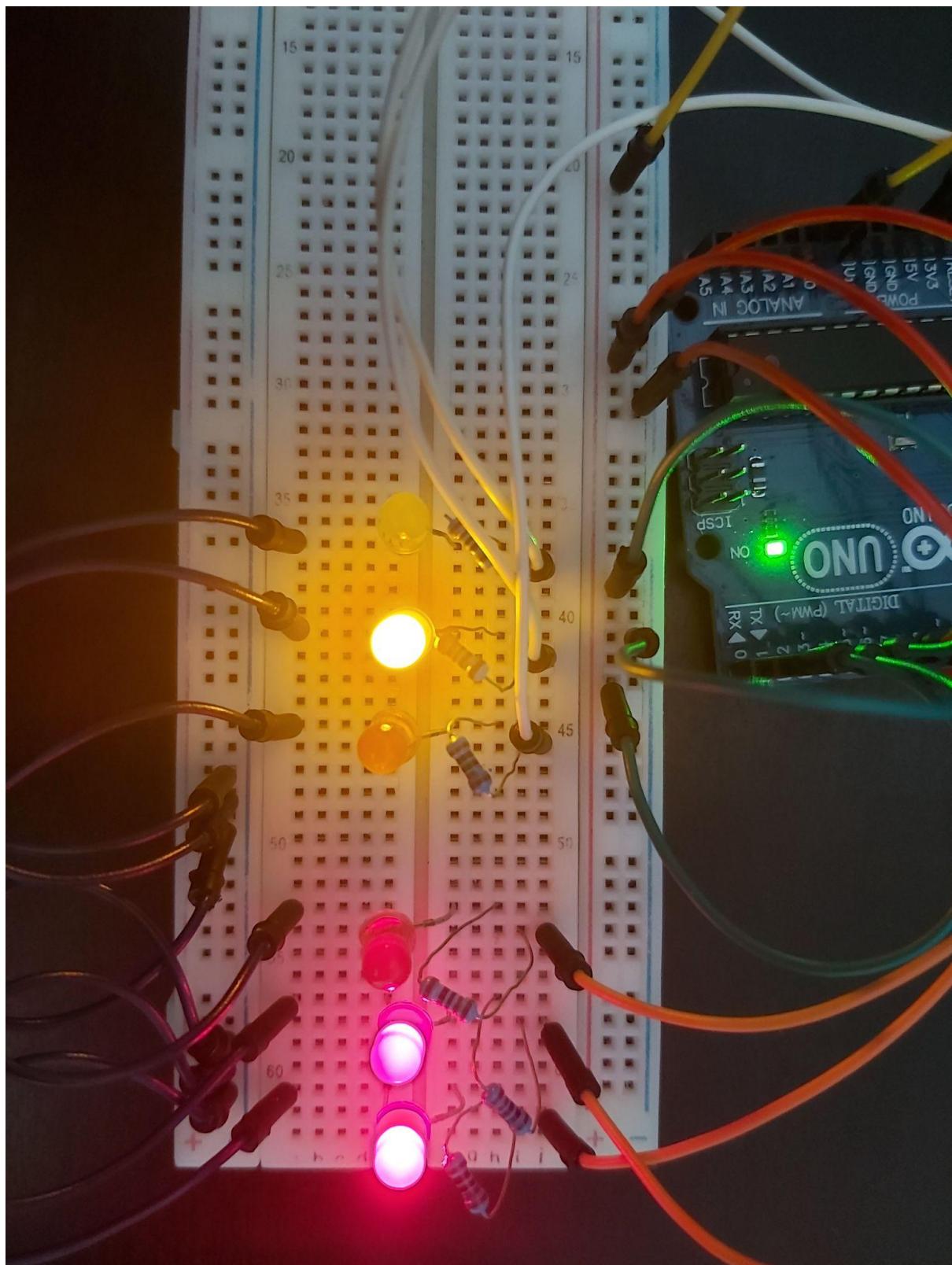


Figura 88 - Testes Subtração: A = 101, B = 010, S = 011, Ov = 1

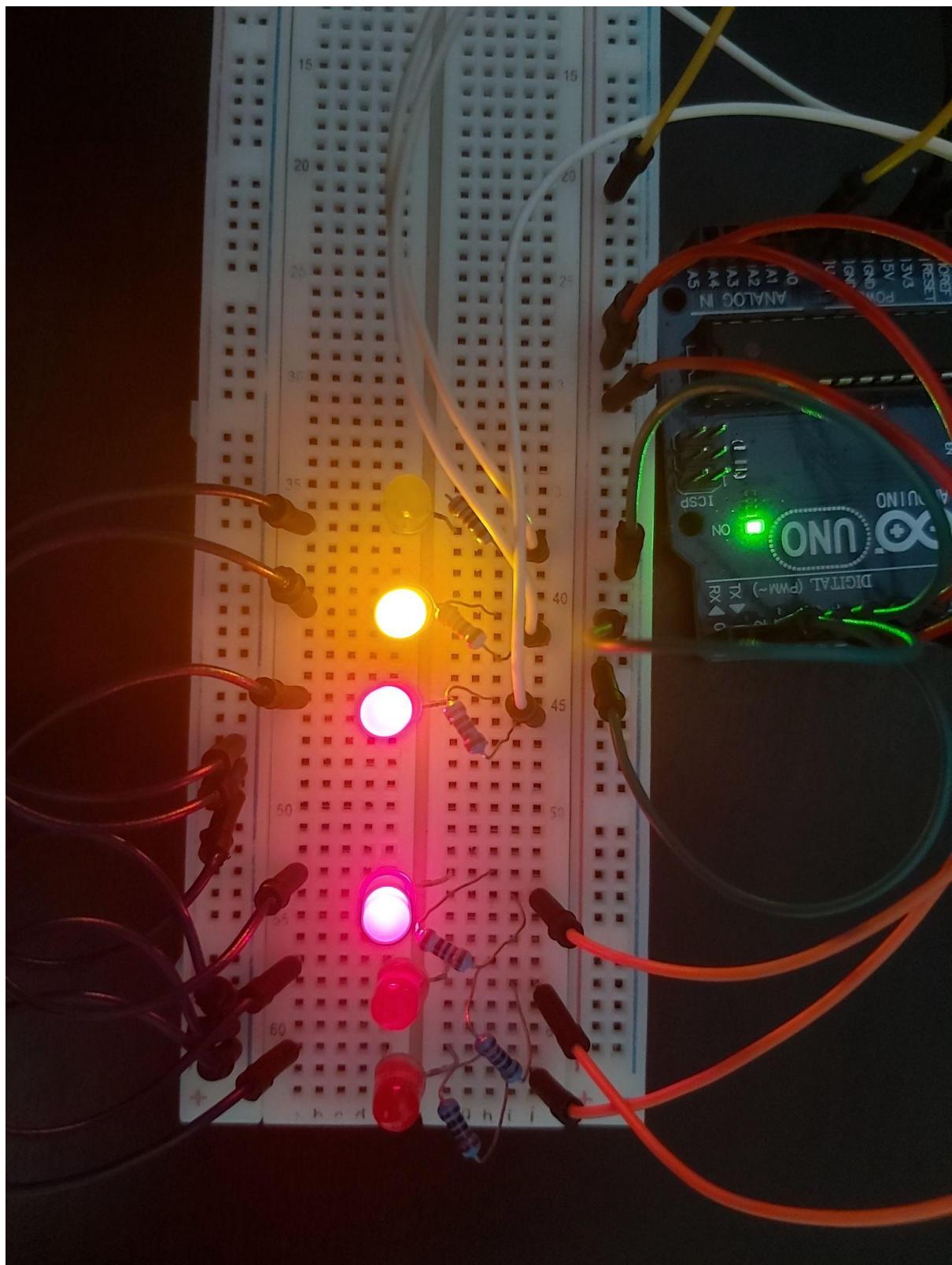


Figura 89 - Testes Subtração: A = 011, B = 111, S = 100, Bw = 1, Ov = 1

3. Exercício 3

Para o exercício final deste primeiro trabalho prático, foi-nos pedido a realização de um contador que, escolhido através de uma variável externa, realizasse a contagem crescente ou decrescente. O conjunto de números que devem estar na saída são o -1, -3, -5 e -7.

Comecemos por definir o modelo geral onde identificamos as entradas, neste caso é apenas o selector, visto que podemos alternar entre os modos de contagem, e as saídas. Como é possível observar, o menor número que existe é o -7, ou seja, um número relativo a quatro bits. Logo, temos quatro saídas. Como referido no enunciado, o facto de ser um contador crescente ou decrescente, significa que existe memória do estado atual para saber qual o próximo estado. Logo, será necessário um CLOCK^[1]. Este será produzido por um botão, quando este é carregado. (Ver Figura 90)

Conseguimos apresentar um modelo Moore-Mealey^[1] mas este ainda é extremamente básico, visto que ainda não obtivemos os detalhes necessários para o desenvolver. (Ver Figura 91)

Dado que existem quatro números desejados nas saídas, temos quatro estados. Assim, se aplicarmos o logaritmo de base dois ao nosso número de estados, é possível concluir que precisamos de dois flip-flop's^[1] para o bloco de memória. Mas, com esta informação, já conseguimos preencher a tabela de verdade do circuito. (Ver Figura 92)

Conseguimos preencher os mapas de Karnaugh para as entradas dos flip-flops e os valores de S1 e S2, dado que é possível observar que S3 e So vão ter sempre o mesmo valor. (Ver Figura 93, Figura 94, Figura 95 e Figura 96)

Também conseguimos preencher um Algorithmic State Machine, também chamado de ASM^[1], com uma variável de entrada a controlar o fluxo do circuito. Esta variável de entrada vai ser o selector. (Ver Figura 97)

De ambas as formas, pelos mapas de Karnaugh e pelo ASM, conseguimos obter as expressões para Do, D1, S1 e S2, e já verificámos que So e S3 são 1. (Ver Figura 98, Figura 99, Figura 100, Figura 101, Figura 102 e Figura 103)

Tal como no segundo exercício, na expressão do D1 temos sempre um número ímpar de números não complementares e as mesmas variáveis presentes. Podemos simplificar a mesma usando o operador lógico XOR. (Ver Figura 104)

Agora, depois de obtermos todas estas informações, conseguimos detalhar melhor o nosso modelo Moore-Mealey. (Ver Figura 105)

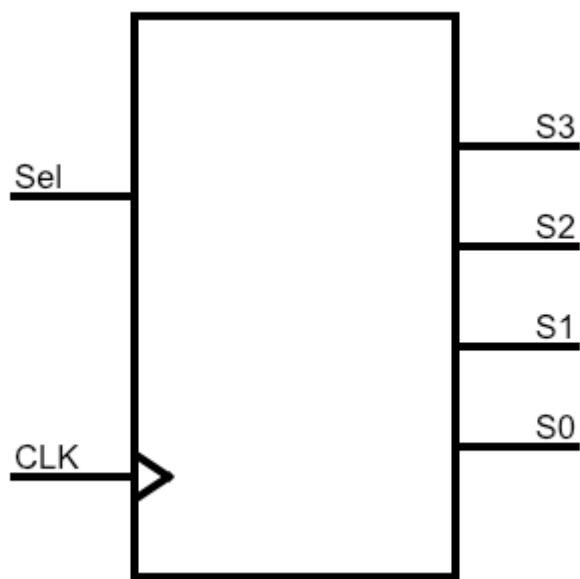


Figura 90 - Modelo Geral

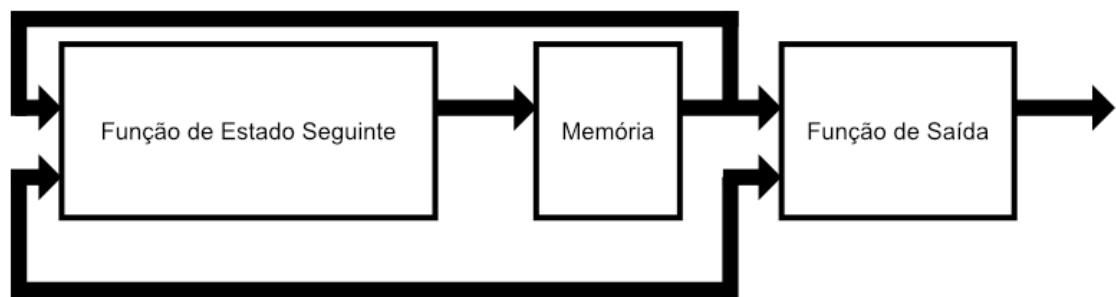


Figura 91 - Modelo Moore-Mealey

Entrada	Estado Presente		Estado Seguinte		Saídas			
E	Q^*		Q		S			
Sel	Q_1^*	Q_0^*	Q_1	Q_0	S_3	S_2	S_1	S_0
0	0	0	0	1	1	0	0	1
0	0	1	1	0	1	0	1	1
0	1	0	1	1	1	1	0	1
0	1	1	0	0	1	1	1	1
1	0	0	1	1	1	0	0	1
1	0	1	0	0	1	0	1	1
1	1	0	0	1	1	1	0	1
1	1	1	1	0	1	1	1	1

Figura 92 - Tabela de Verdade

D0

$\overline{Q0^*}$

Sel |

1	0	0	1
1	0	0	1

$\overline{Q1^*}$

Figura 93 - Mapa de Karnaugh de Do

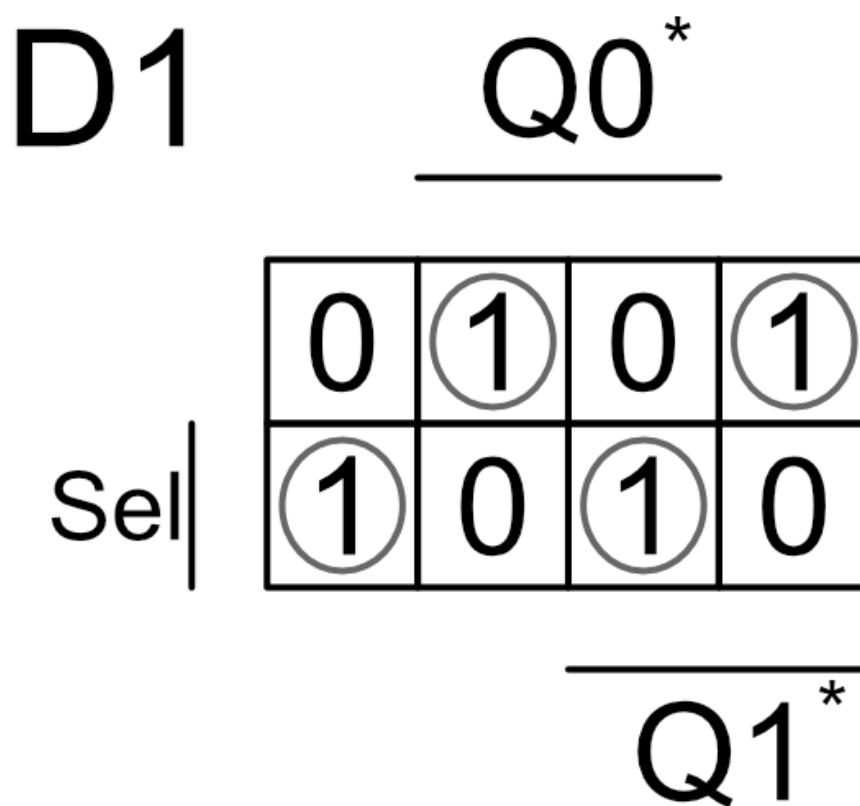


Figura 94 - Mapa de Karnaugh de D1

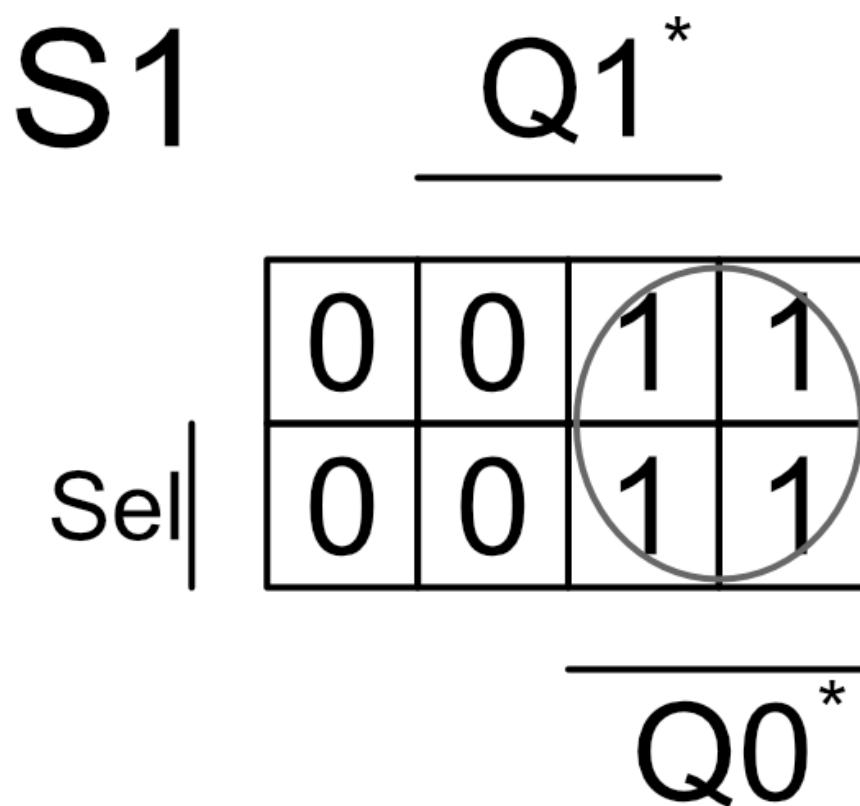


Figura 95 - Mapa de Karnaugh de S1

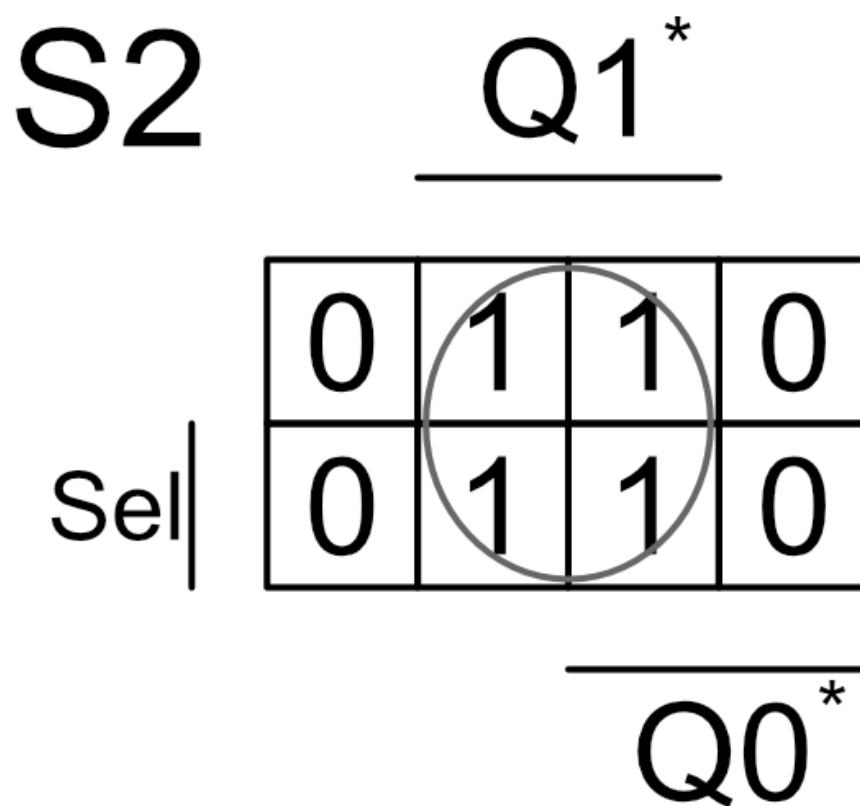


Figura 96 - Mapa de Karnaugh de $S2$

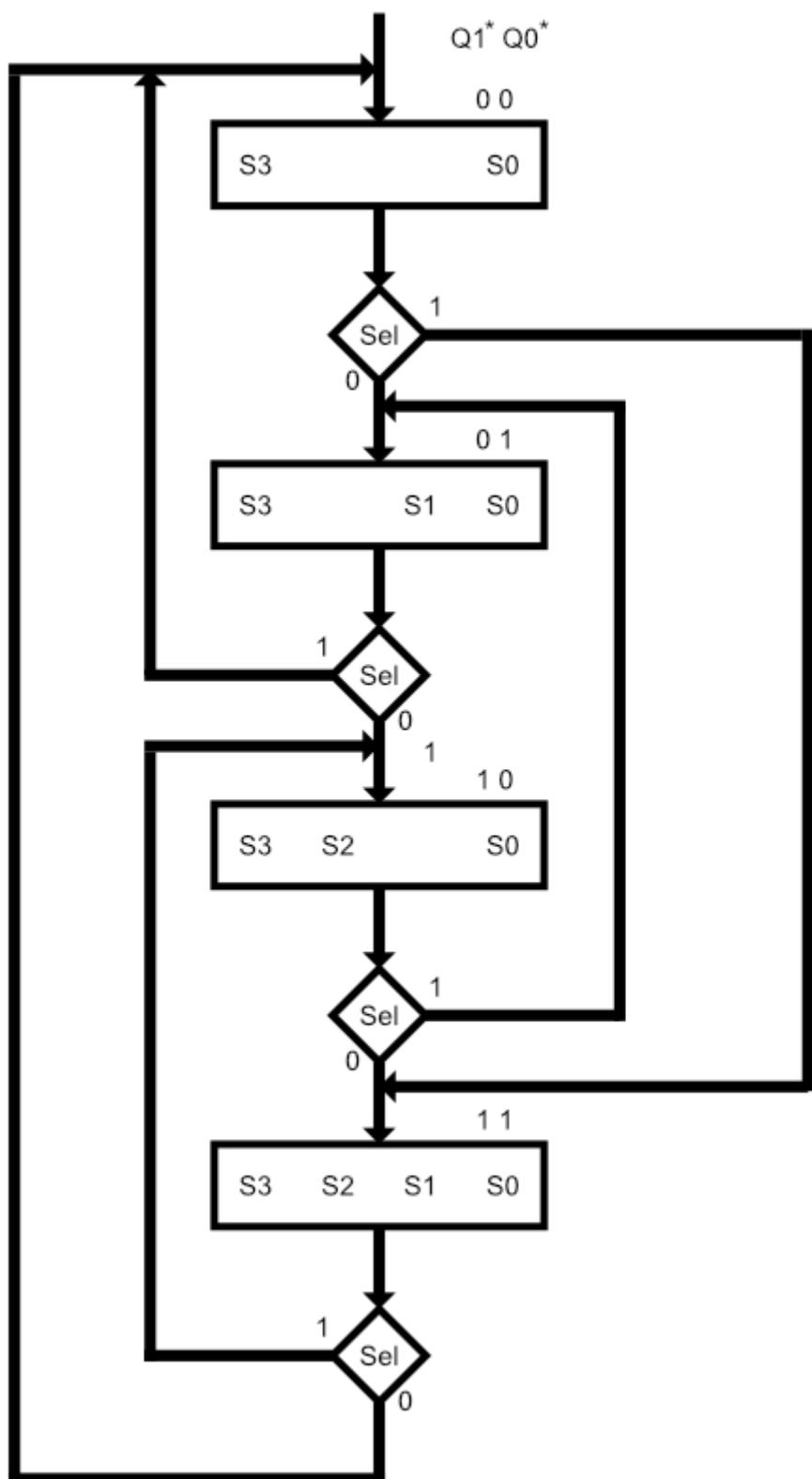


Figura 97 - Algorithmic State Machine, ASM

$$\overline{D_0} = \overline{Q_0}^*$$

Figura 98 - Expressão de D0

$$D_1 = \overline{\text{Sel.Q0}}^* \overline{\text{Q1}}^* + \overline{\text{Sel.Q0}}^* \overline{\text{Q1}} + \overline{\text{Sel.Q0}}^* \overline{\text{Q1}}^* + \text{Sel.Q0}^* \text{Q1}^*$$

Figura 99 - Expressão de D1

$$S_1 = Q_0^*$$

Figura 100 - Expressão de S1

$$S_2 = Q_1^*$$

Figura 101 - Expressão de S2

$$S_0 = 1$$

Figura 102 - Expressão de S0

$$S_3 = 1$$

Figura 103 - Expressão de S3

$$D_1 = Sel \wedge Q_0^* \wedge Q_1^*$$

Figura 104 - Expressão Simplificada de D1

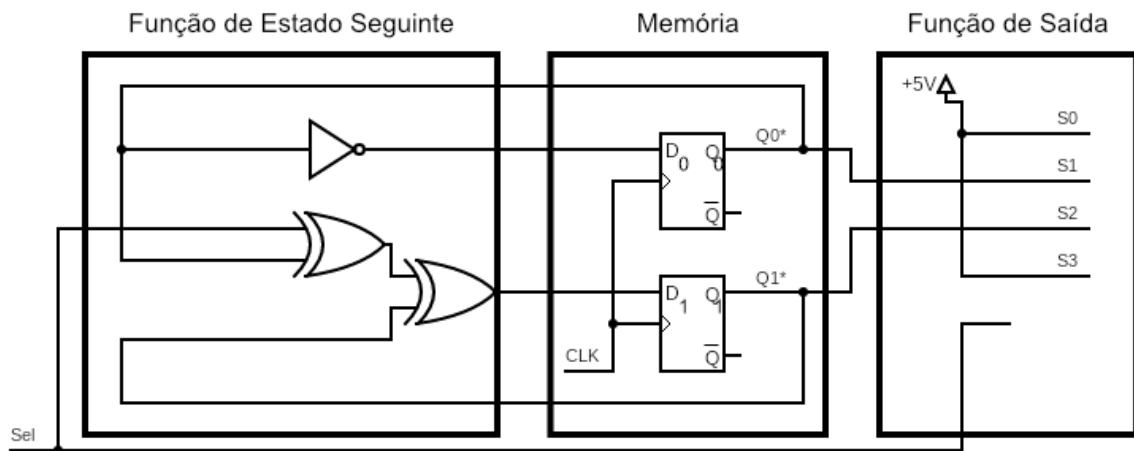


Figura 105 - Modelo Moore-Mealey Detalhado

Código

Começamos por definir os pinos. Como entrada temos apenas o Selector. Para as saídas temos os bits representantes do resultado do contador, logo quatro saídas S₃, S₂, S₁ e So. (Ver Figura 106)

```
#define pinSel 3  
  
#define pinS0 4  
#define pinS1 5  
#define pinS2 6  
#define pinS3 7
```

Figura 106 - Definição de Pinos

Definimos as variáveis a serem usadas, neste caso dado que só podem tomar valores de 0 ou 1, estas serão booleanas. Para este exercício, estas são o Selector, Sel, uma variável para cada bit de S, S₃, S₂, S₁, So. Vamos também precisar de uma variável para cada entrada dos *flip-flop's*, D₁ e D₀. (Ver Figura 107)

```
bool Sel, D0, D1, S0, S1, S2, S3;  
volatile bool Q0, Q1;
```

Figura 107 - Definição de Variáveis

É implementado o *flip-flop* D que, como previamente observado, os valores que produz são a identidade do D. (Ver Figura 108)

```
boolean flip_flop_D(boolean D)  
{  
    return D;  
}
```

Figura 108 - Implementação de flip_flop_D

Definimos a função de estado seguinte que vai afectar os valores de entrada dos *flip-flop's*, ou seja, D₁ e D₀, pelas expressões apontadas anteriormente. (Ver Figura 109)

```
void funcaoEstadoSeguinte()  
{  
    D1 = Sel ^ Q0 ^ Q1;  
    D0 = !Q0;  
}
```

Figura 109 - Implementação de funcaoEstadoSeguinte

A função de saída afecta apenas cada S, mas já sabemos que o primeiro e o último, S₃ e S₀, respectivamente, são sempre 1 e que S₂ e S₁ tomam os valores de Q₁ e Q₀, respectivamente. (*Ver Figura 110*)

```
void funcaoSaida()
{
    S0 = true;
    S1 = Q0;
    S2 = Q1;
    S3 = true;
}
```

Figura 110 - Implementação de funçãoSaida

A função que permite ao estado seguinte tornar-se no estado presente é o CLK. Este atualiza os valores de Q₁ e Q₀ sempre que é premido o botão. (*Ver Figura 111*)

```
void CLK()
{
    Q0 = flip_flop_D(D0);
    Q1 = flip_flop_D(D1);
}
```

Figura 111 - Implementação de CLK

Criamos uma função que obtém o valor do selector. (*Ver Figura 112*)

```
void readInputs()
{
    Sel = digitalRead(pinSel);
}
```

Figura 112 - Implementação de readInputs

E criamos uma função que escreve os valores de S nos pinos apropriados. (*Ver Figura 113*)

```
void writeOutputs()
{
    digitalWrite(pinS0, true);
    digitalWrite(pinS1, S1);
    digitalWrite(pinS2, S2);
    digitalWrite(pinS3, true);
}
```

Figura 113 - Implementação de writeOutputs

No *setup* definimos o modo dos pinos, assim como a função a ser chamada quando existe um tensão ascendente no pino do CLK e ligamos as interrupções. (Ver Figura 114)

```
void setup() {
    pinMode(pinSel, INPUT);
    pinMode(pinS0, OUTPUT);
    pinMode(pinS1, OUTPUT);
    pinMode(pinS2, OUTPUT);
    pinMode(pinS3, OUTPUT);
    pinMode(8, OUTPUT);

    attachInterrupt(0, CLK, RISING);
    interrupts();
}
```

Figura 114 - Setup

Por fim, temos o *loop* do programa. Este, lê o valor do selector, obtém o valor do estado seguinte, obtém os valores para as saídas e, por fim, escreve-as Sempre que o utilizador carrega no botão, a execução do *loop* é suspensa e uma nova thread é corrida. Neste caso, esta thread possui apenas o CLK. Quando finalizado, retorna para o ponto onde a sua execução foi suspensa e esta é resumida. (Ver Figura 115)

```
void loop() {
    readInputs();
    funcaoEstadoSeguinte();
    funcaoSaida();
    writeOutputs();
}
```

Figura 115 - Loop

Durante os testes, foram observados alguns problemas quando o botão era premido. Podia saltar estados, voltar atrás nos mesmos ou até mesmo permanecer no mesmo. Isto é devido ao *debounce* que os botões possuem. Para arranjar este problema e possuirmos uma execução correcta, foi implementado um cronómetro que impede vários cliques de uma vez. Foi necessário criar duas variáveis para o controlo do tempo e modificarmos a função do CLK. (Ver Figura 116 e Figura 117)

```
unsigned long now, ago;
```

Figura 116 - Variáveis de Controlo de Tempo do Botão

```
void CLK()
{
    now = millis();

    if(now - ago >= 200)
    {
        Q0 = flip_flop_D(D0);
        Q1 = flip_flop_D(D1);
        ago = now;
    }
}
```

Figura 117 - CLK Alterado para o seu Correcto Funcionamento com Botão

Foi também testado com a função de geração automática de ondas quadradas que se encontra explicitada nos slides com a matéria dada. Foram necessárias pequenas alterações no *loop* e na montagem do circuito. (*Ver Figura 118 e Figura 119*)

```
void clockGenerator(byte pino, float f)
{
    static boolean state = LOW;
    static unsigned long t1, t0;
    t1 = millis();

    if (t1 - t0 >= 500. / f)
    {
        digitalWrite(pino, state = !state);
        t0 = t1;
    }
}
```

Figura 118 - Implementação de clockGenerator

```
void loop() {
    readInputs();
    funcaoEstadoSeguinte();
    funcaoSaida();
    writeOutputs();
    clockGenerator(8, 1);
}
```

Figura 119 - Alteração do Loop para CLK Automático

Montagem

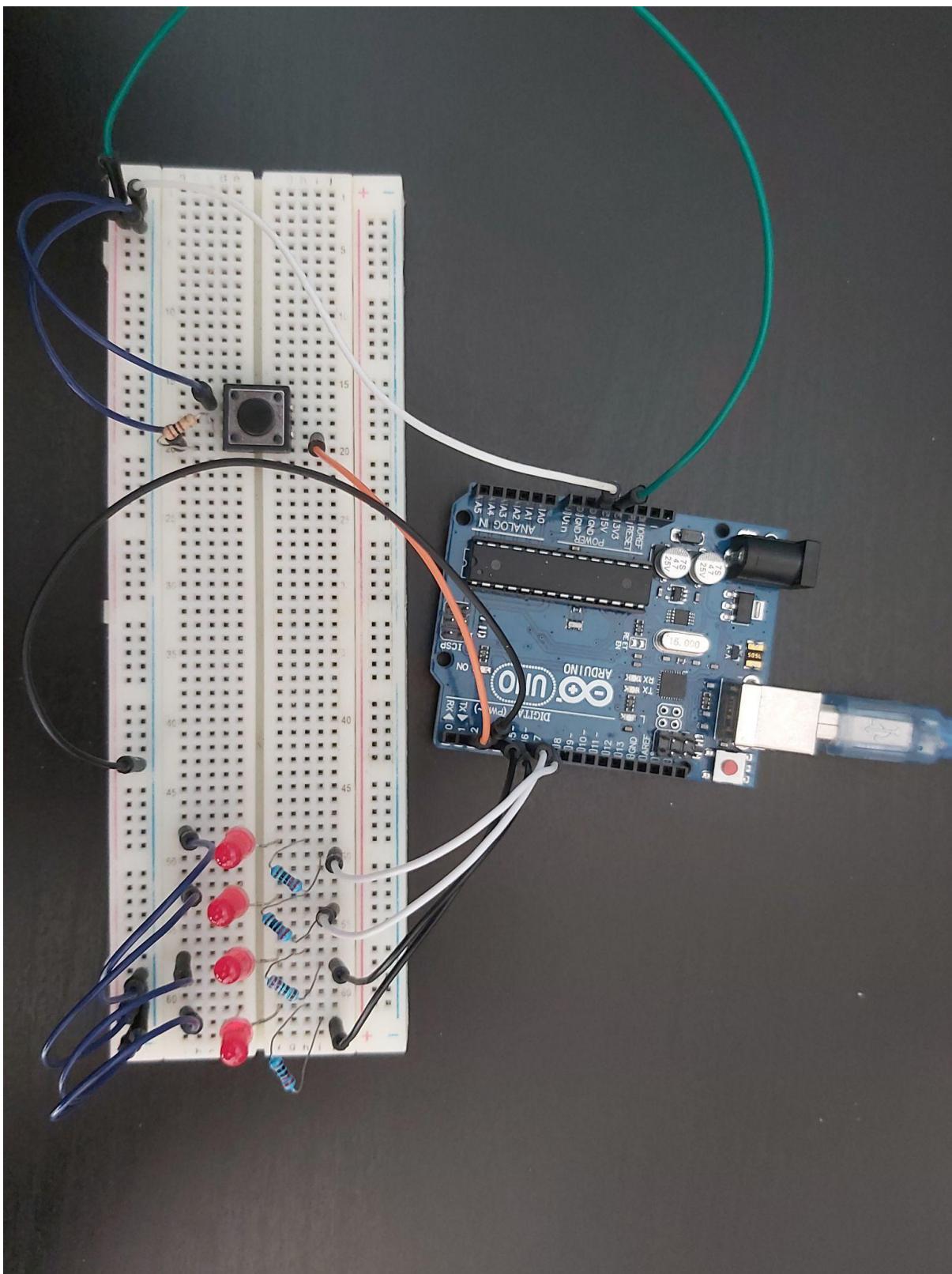


Figura 120 - Montagem Geral

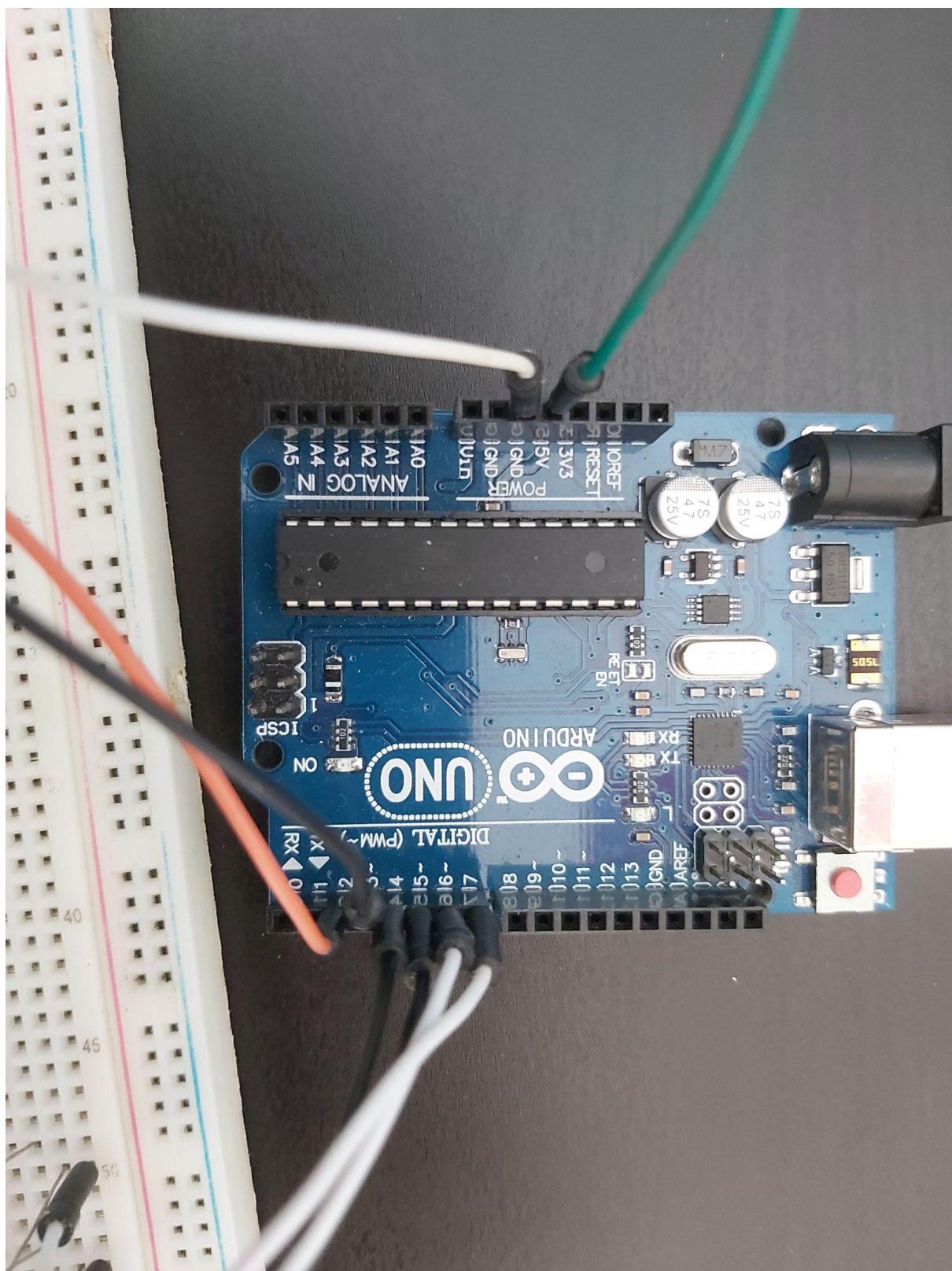


Figura 121 - Montagem Arduíno 1

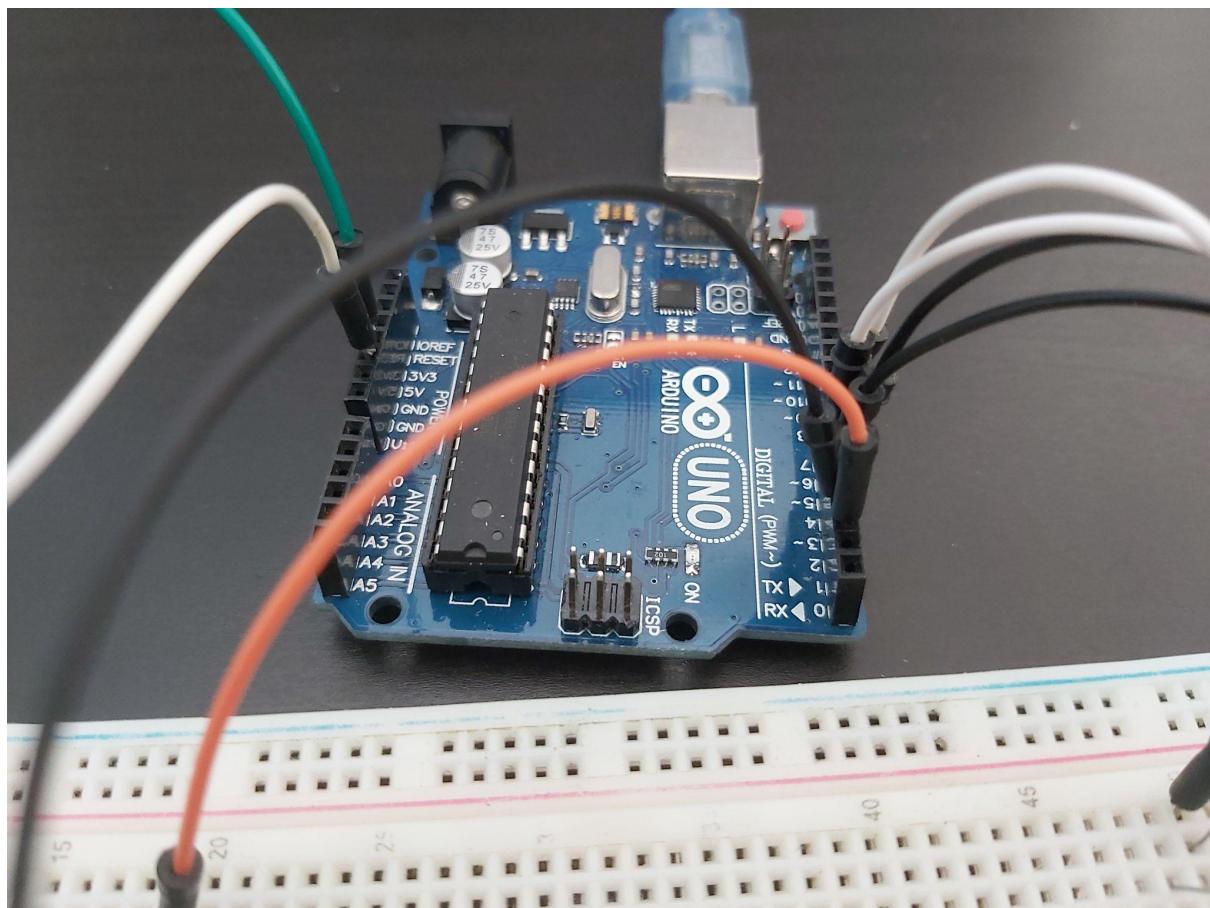


Figura 122 - Montagem Arduíno 122

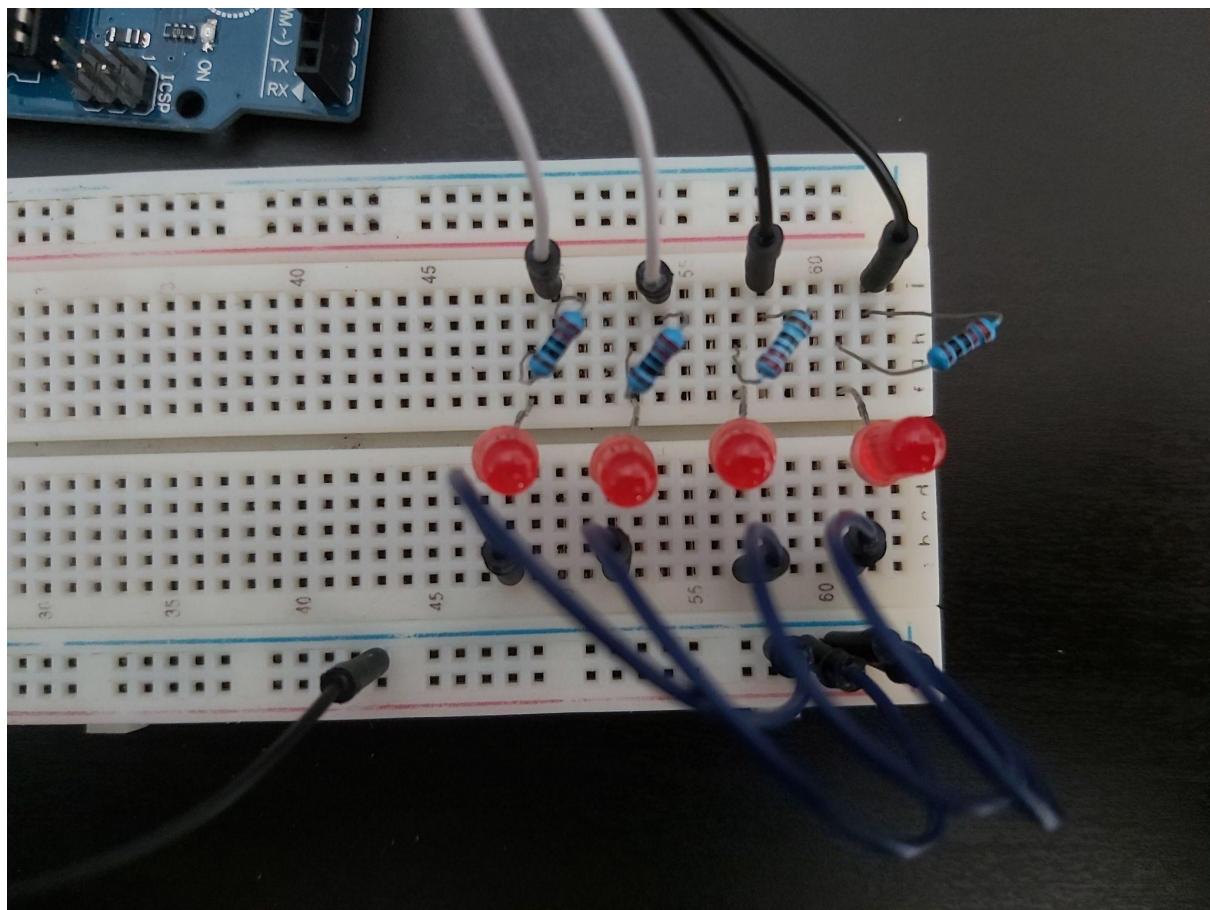


Figura 123 - Montagem Saídas 1

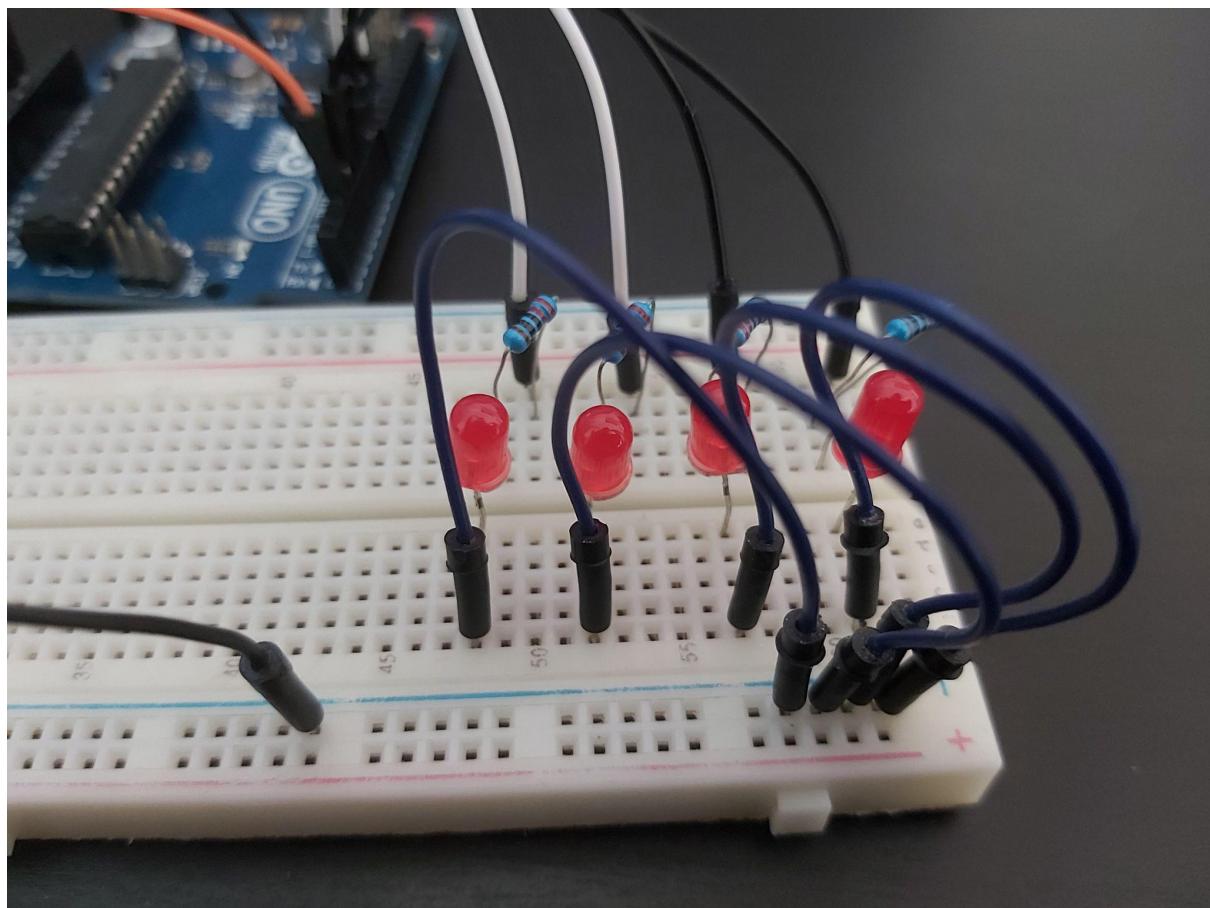


Figura 124 - Montagem Saídas 2

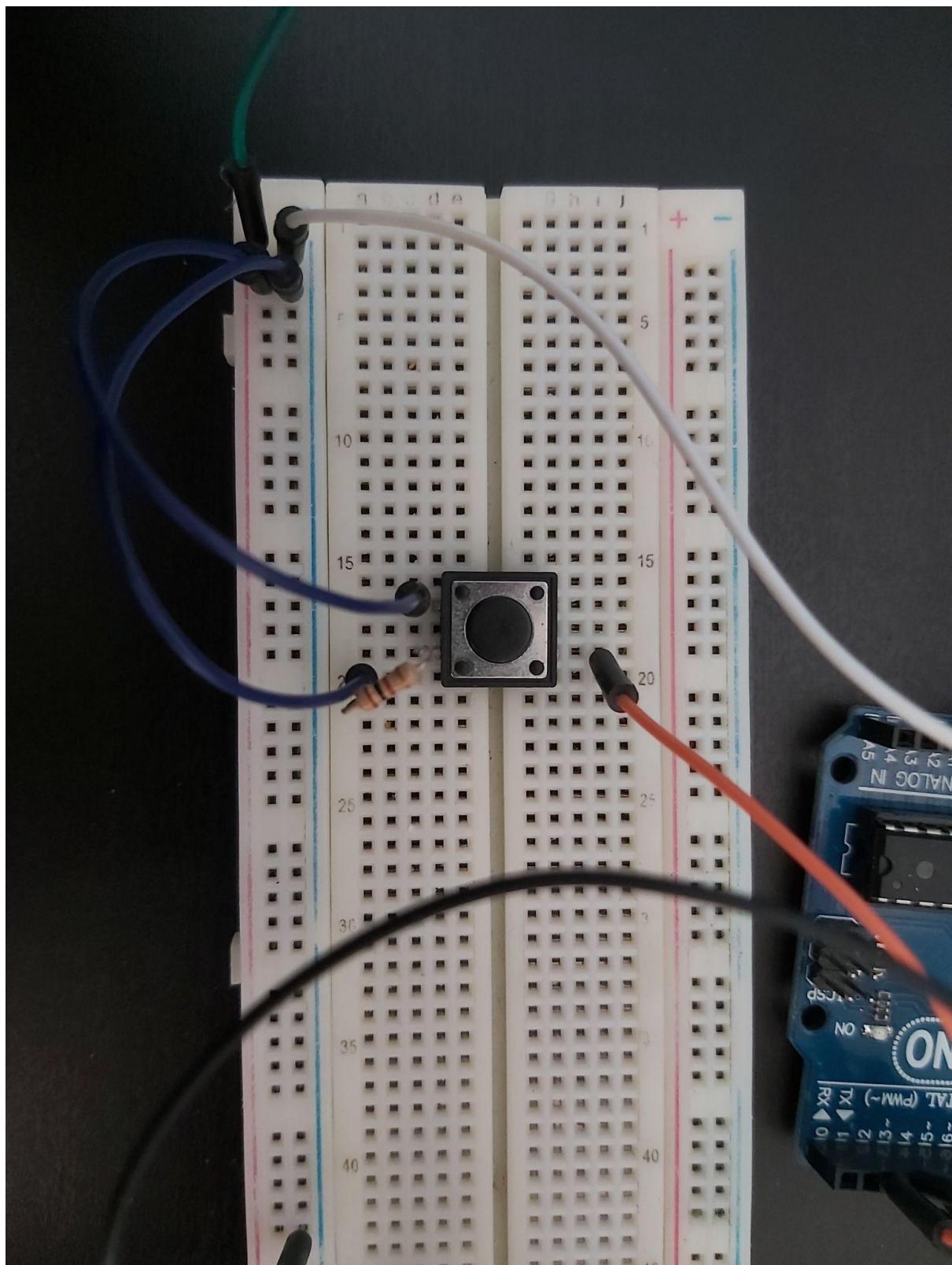


Figura 125 - Montagem Botão 1

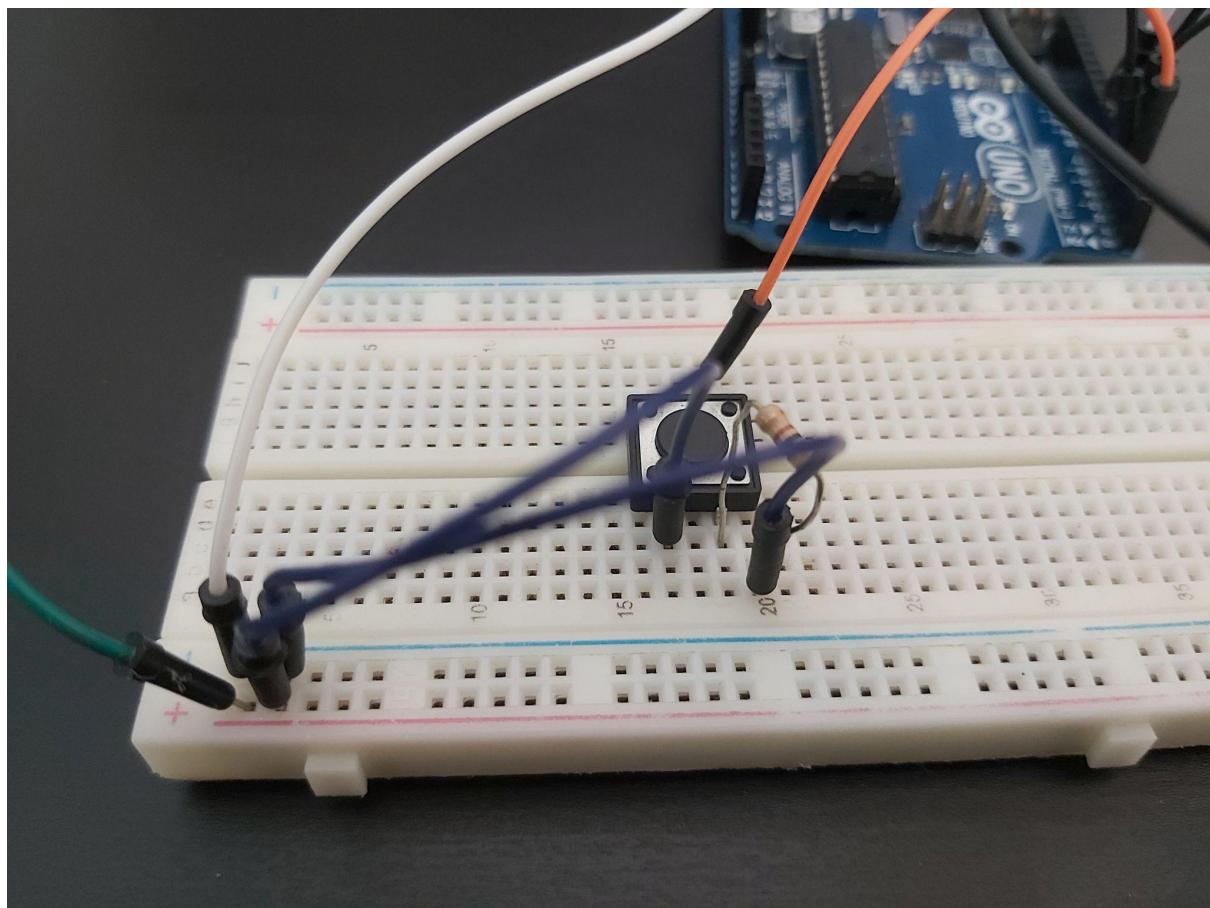


Figura 126 - Montagem Botão 2

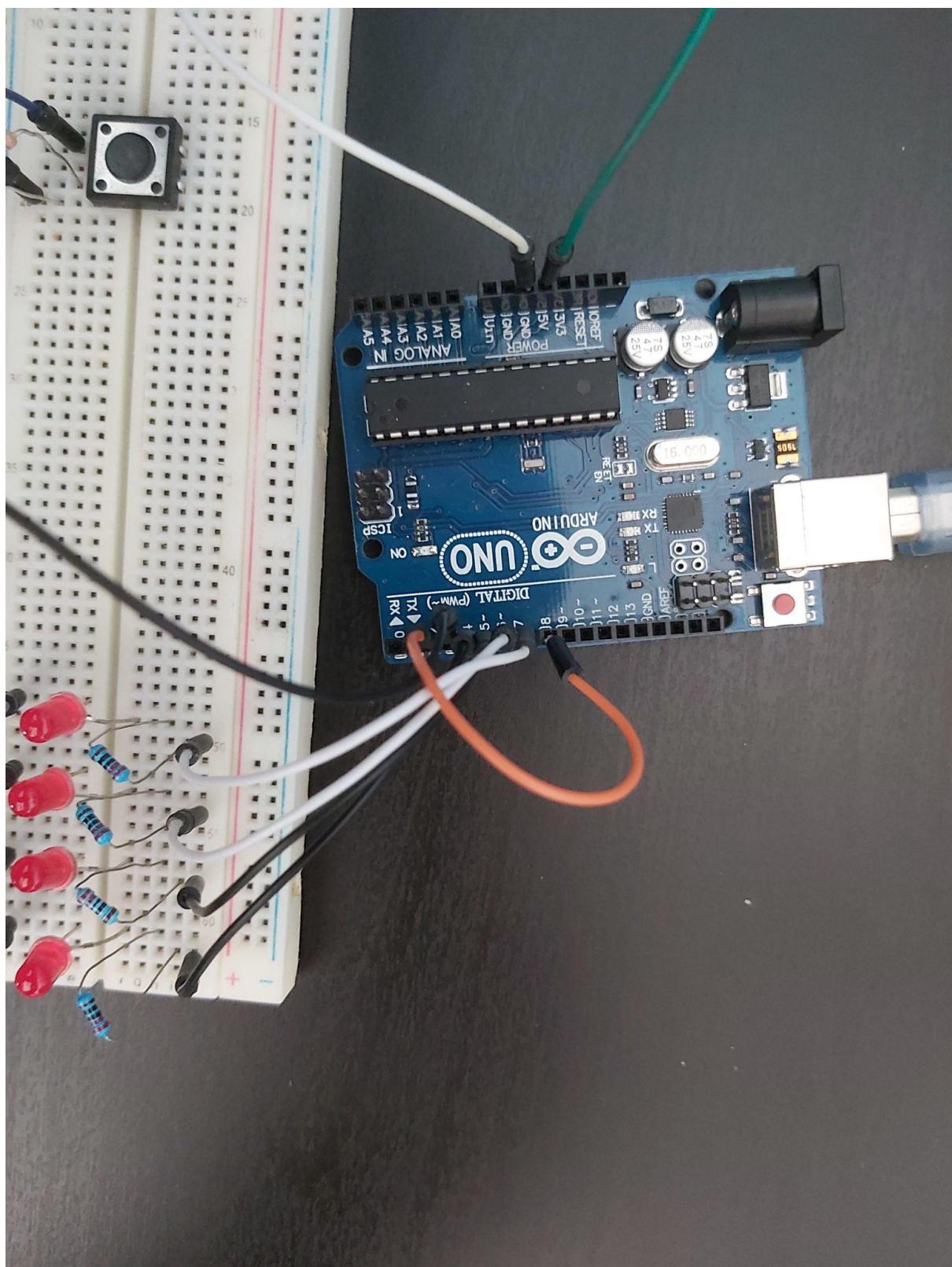


Figura 127 - Montagem CLK Automático 1

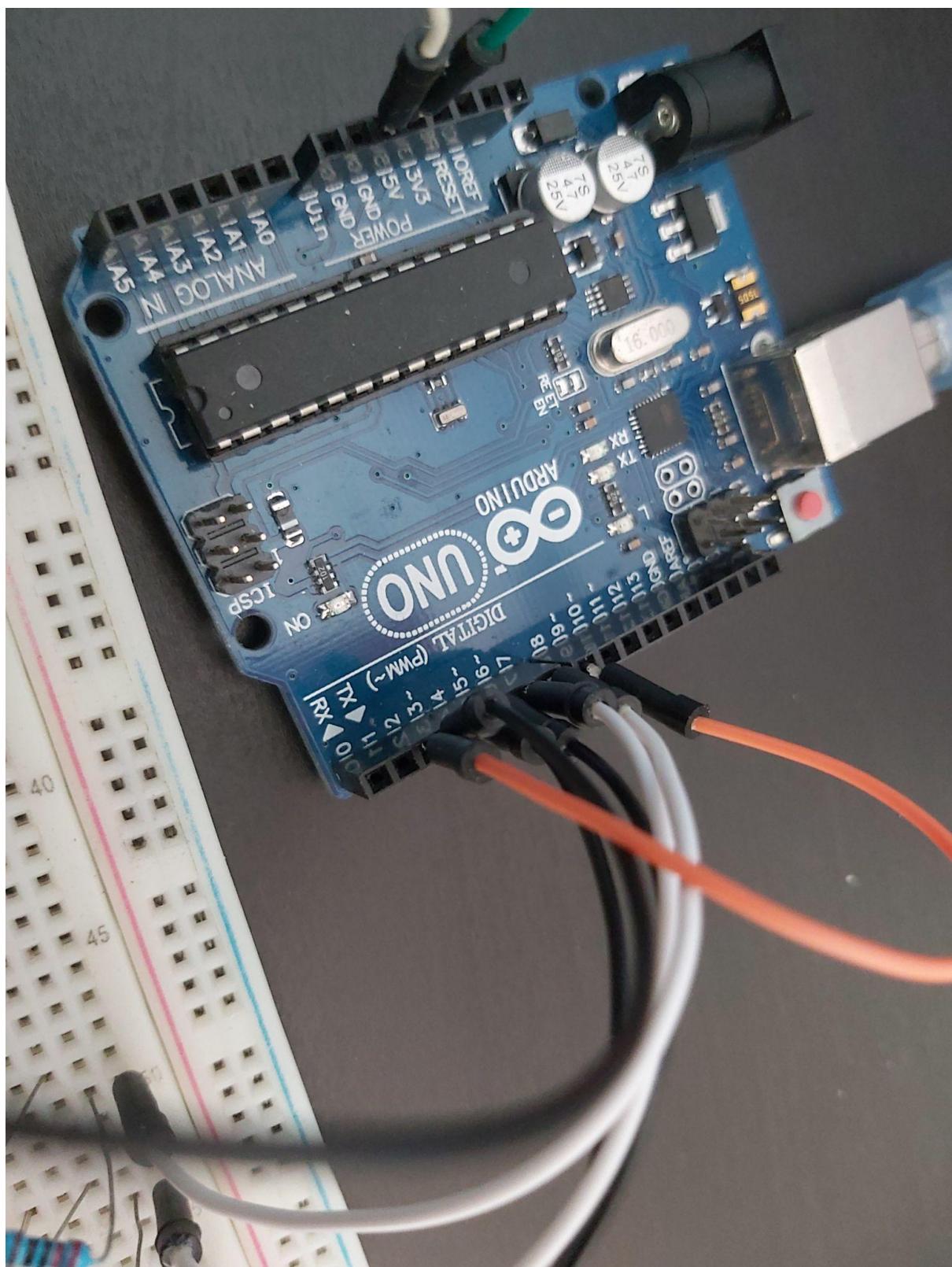


Figura 128 - Montagem CLK Automático 2

Testes

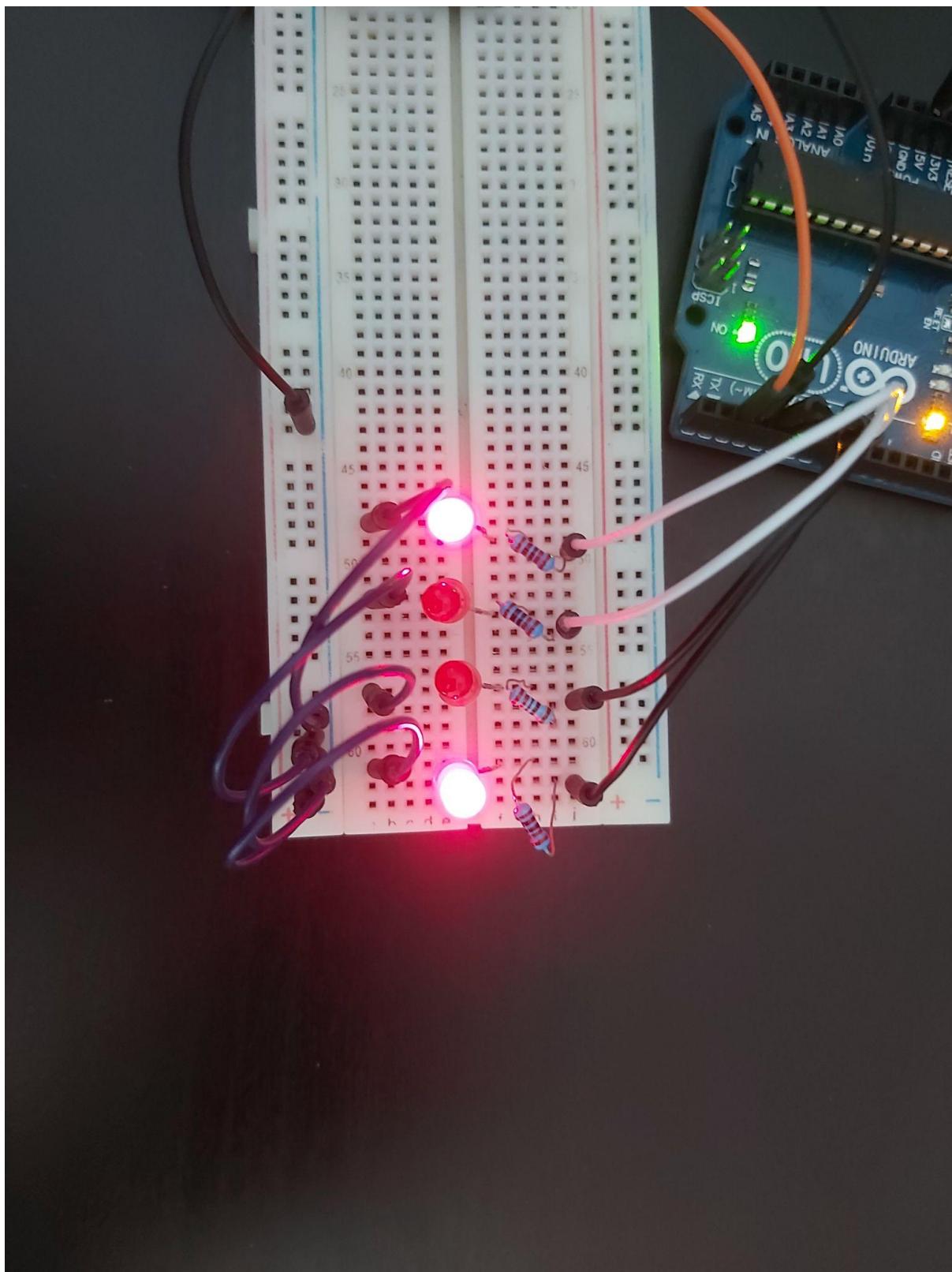


Figura 129 - Testes Contador Crescente: $S = 1001$

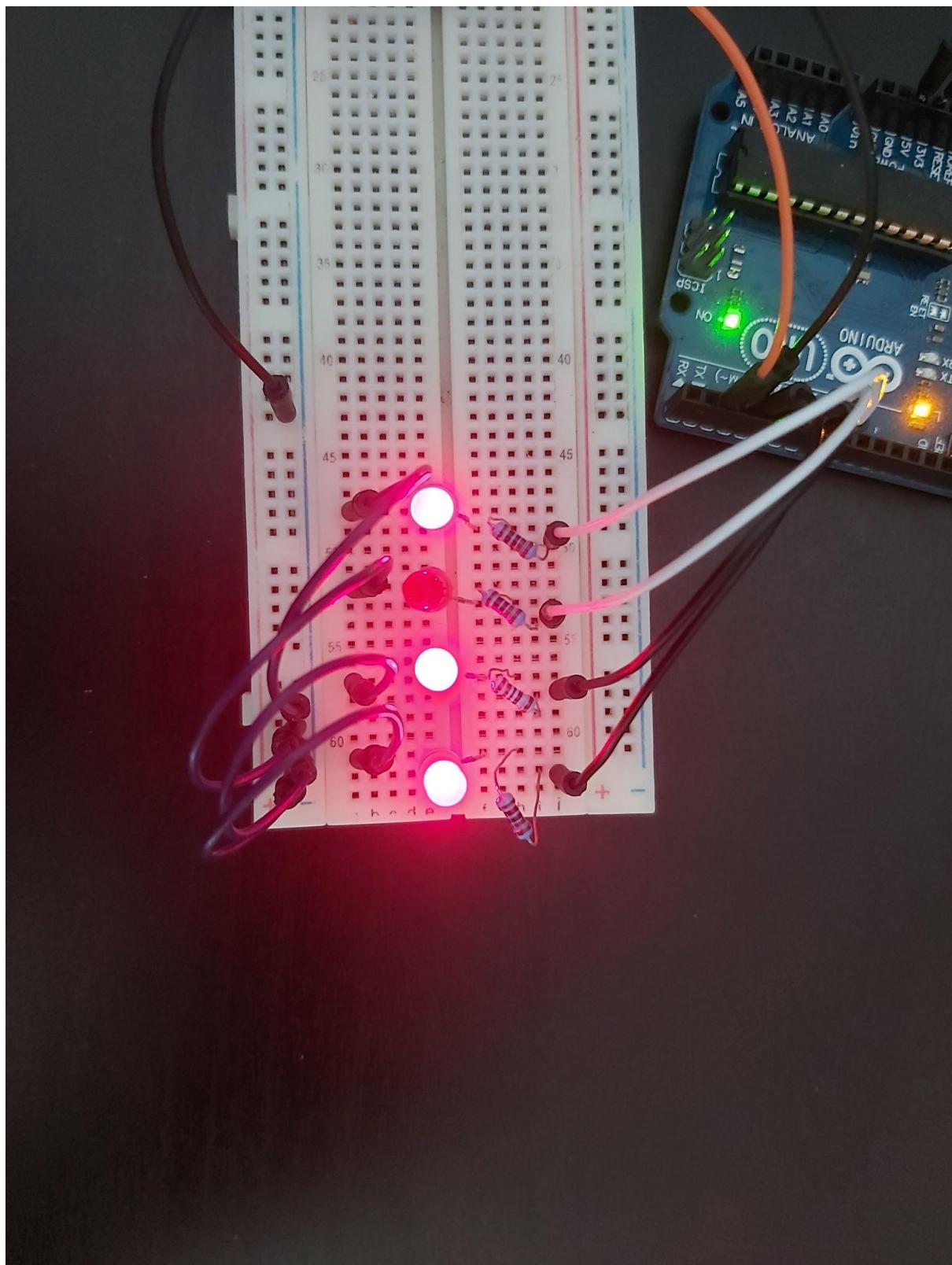


Figura 130 - Testes Contador Crescente: $S = 1011$

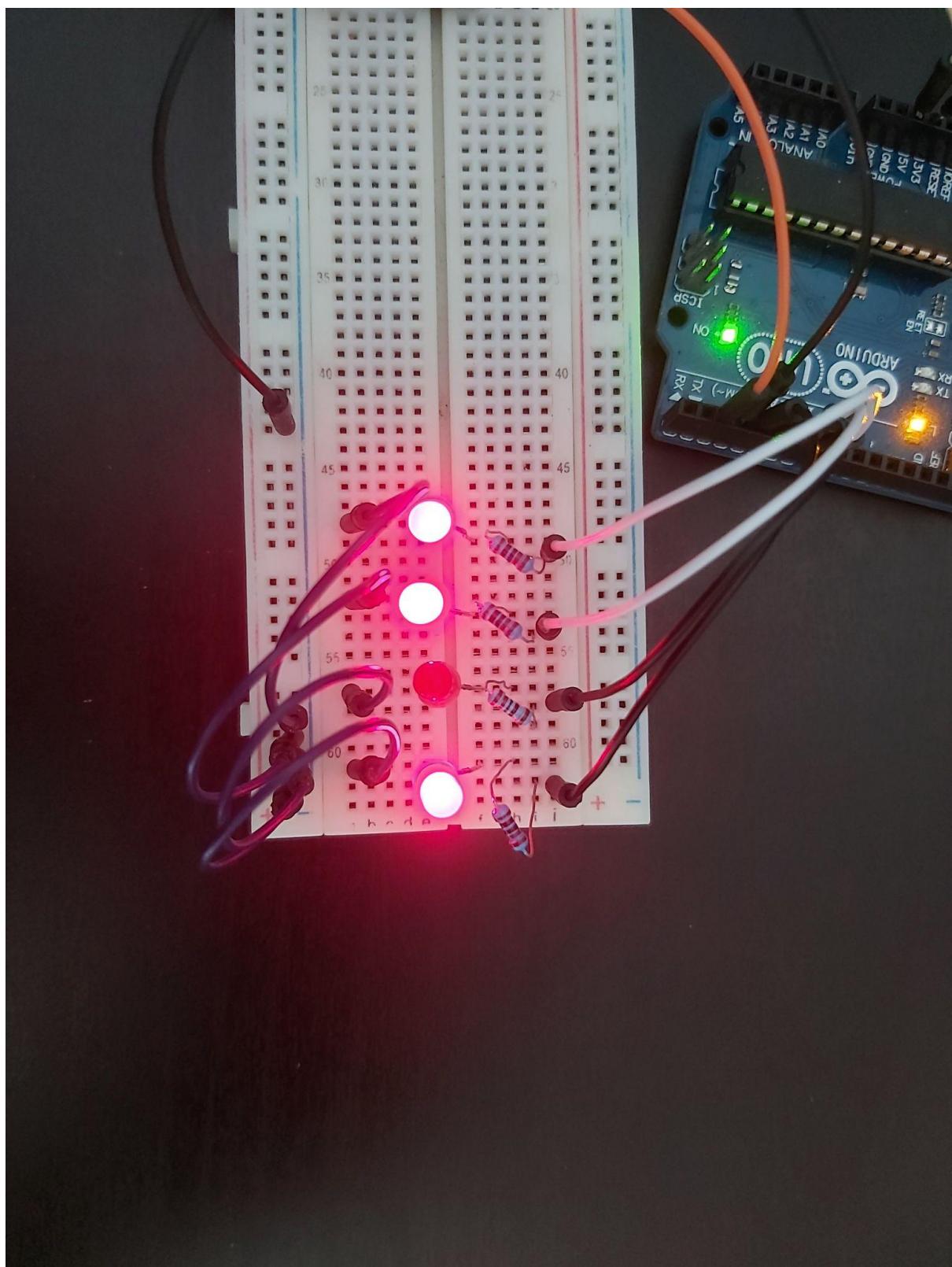


Figura 131 - Testes Contador Crescente: $S = 1101$

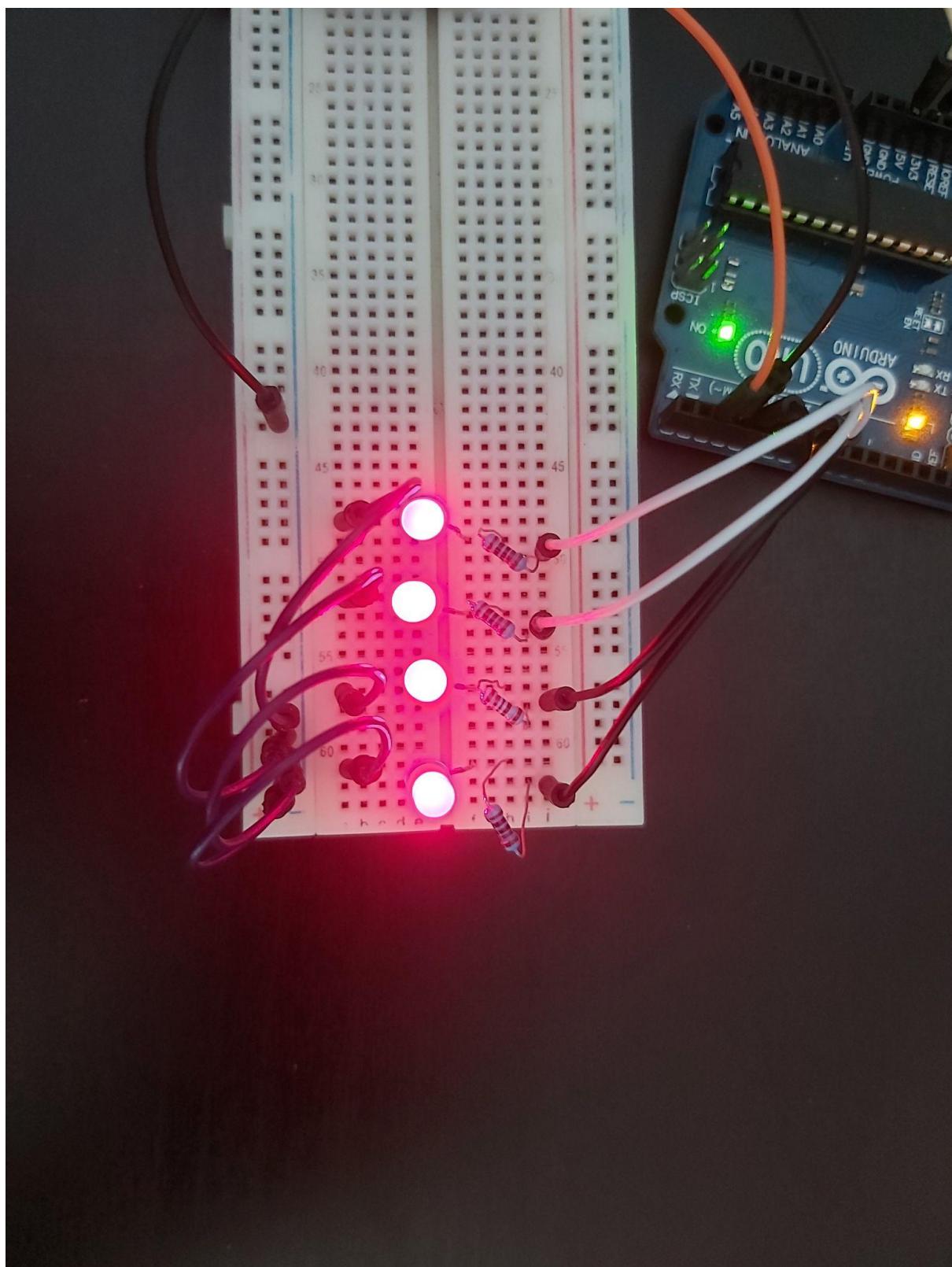


Figura 132 - Testes Contador Crescente: S = 1111

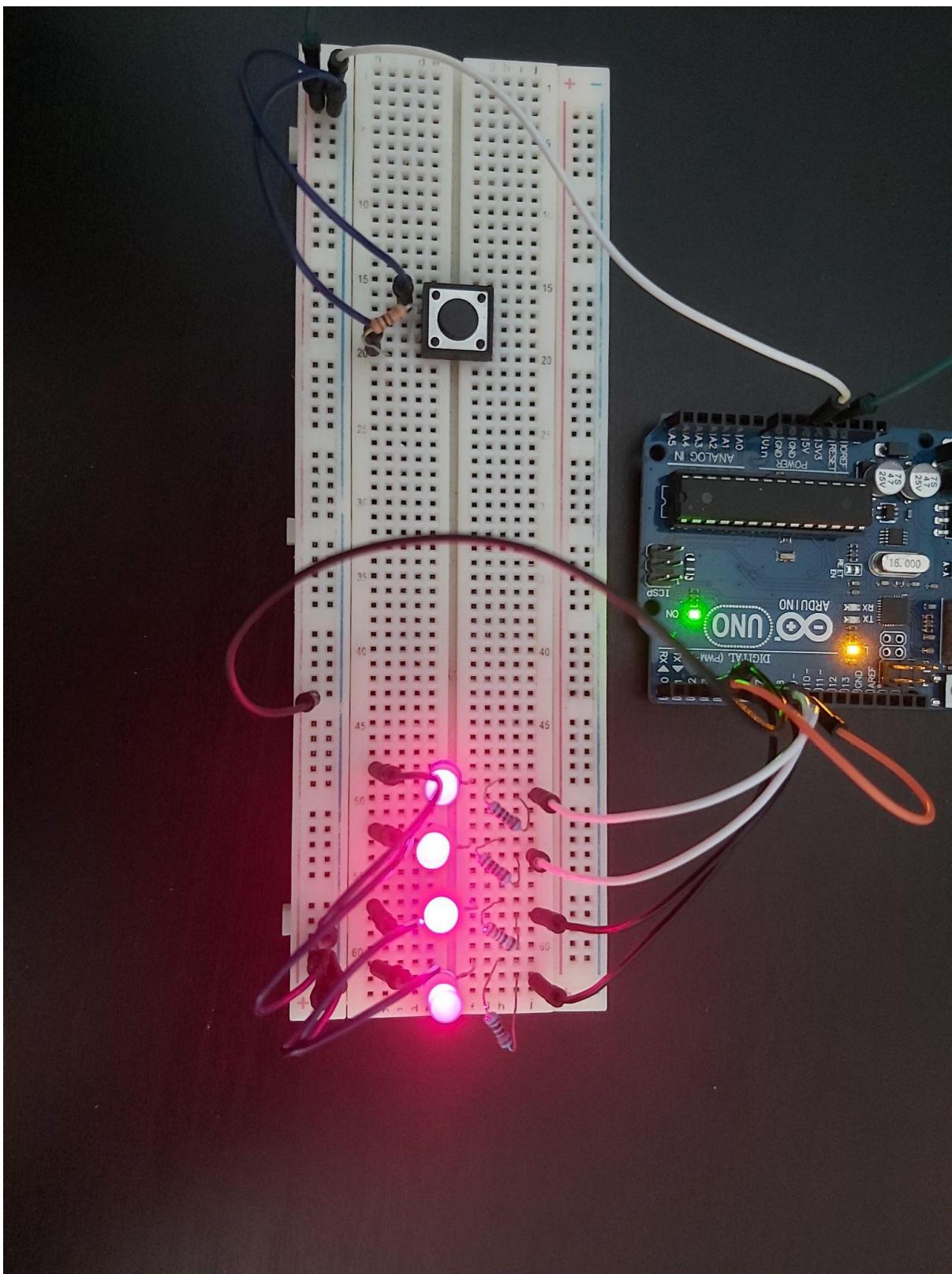


Figura 133 - Testes Contador Decrescente: S = 1111

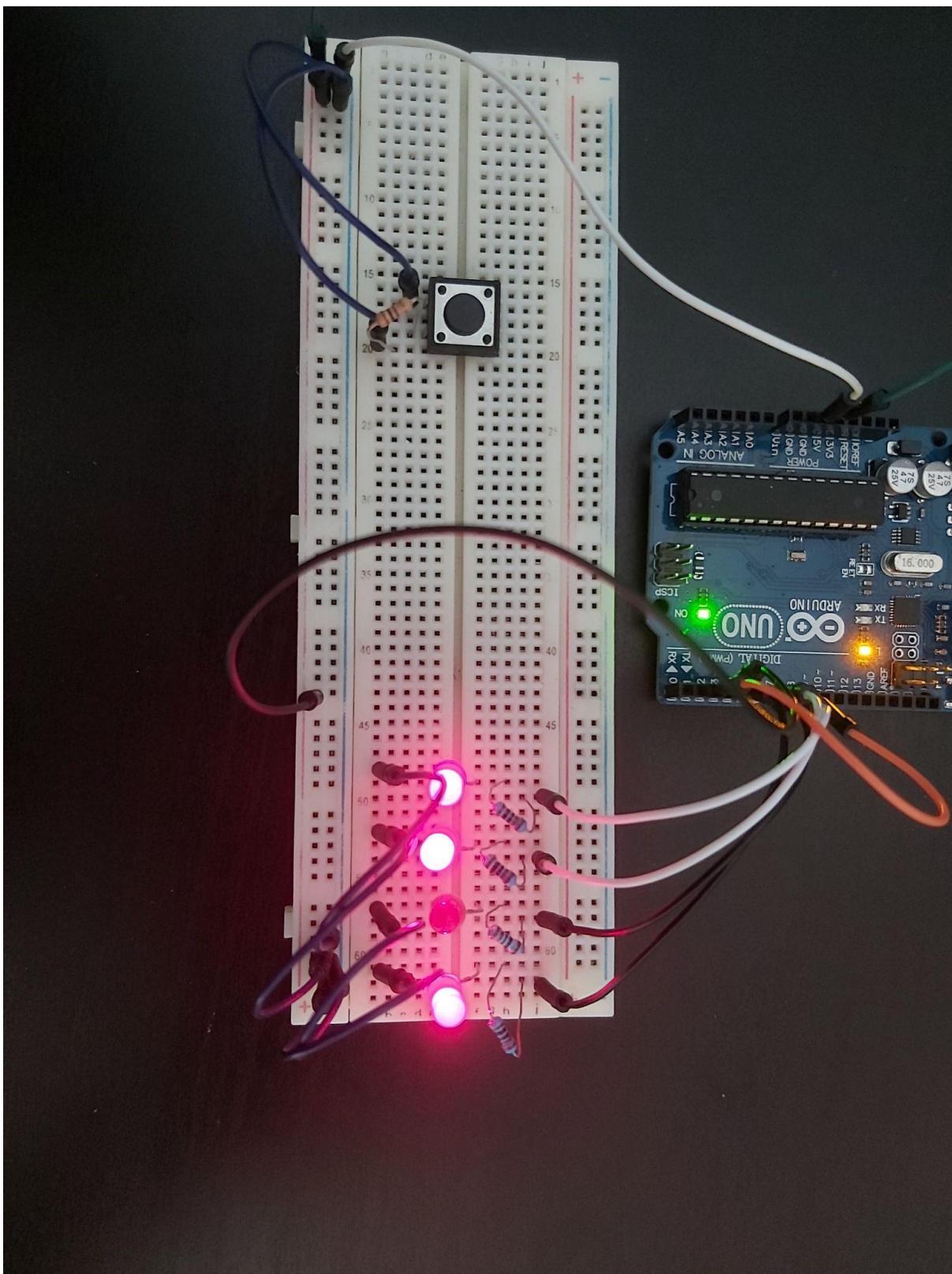


Figura 134 - Testes Contador Decrescente: S = 1101

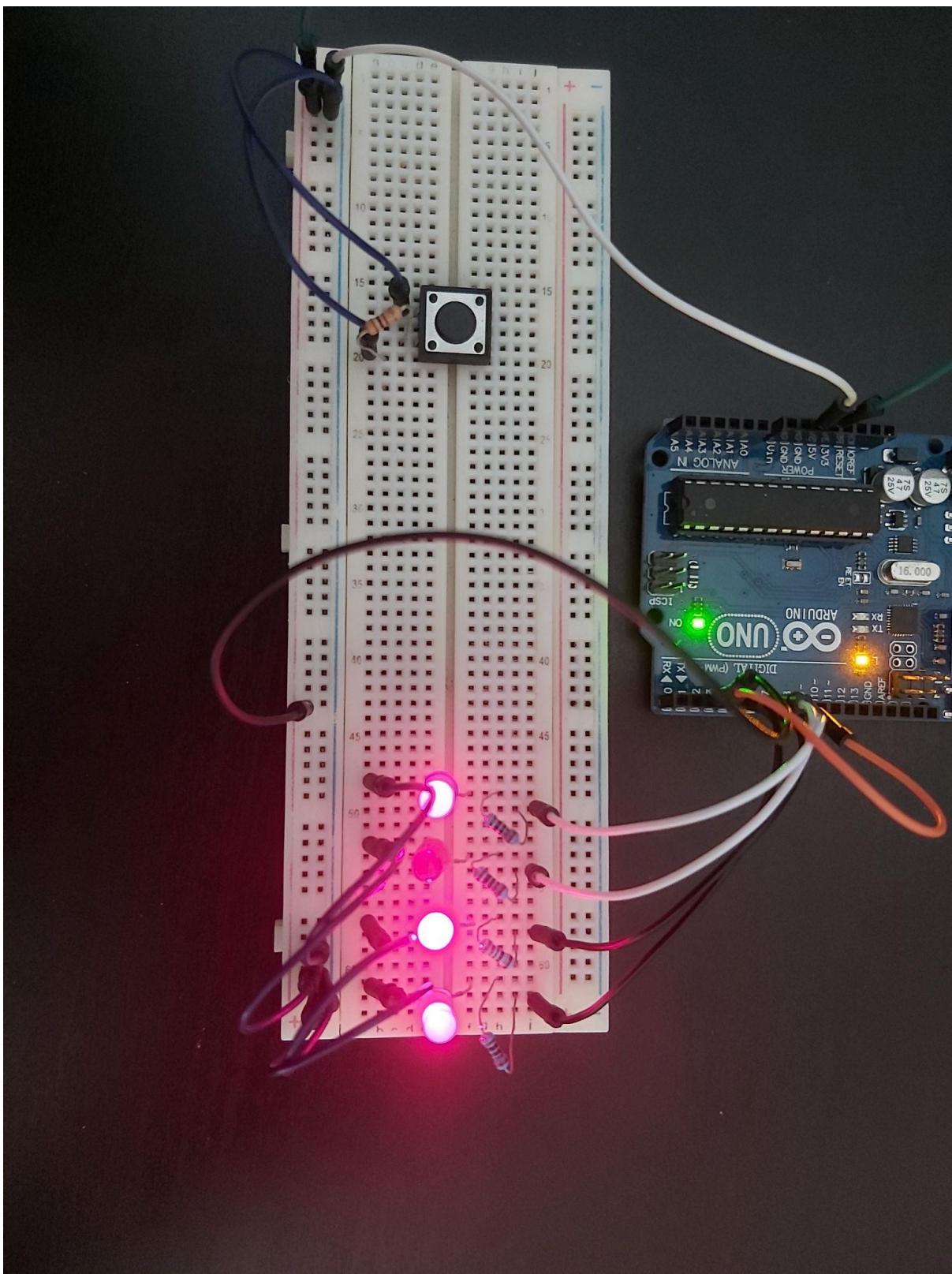


Figura 135 - Testes Contador Decrescente: S = 1011

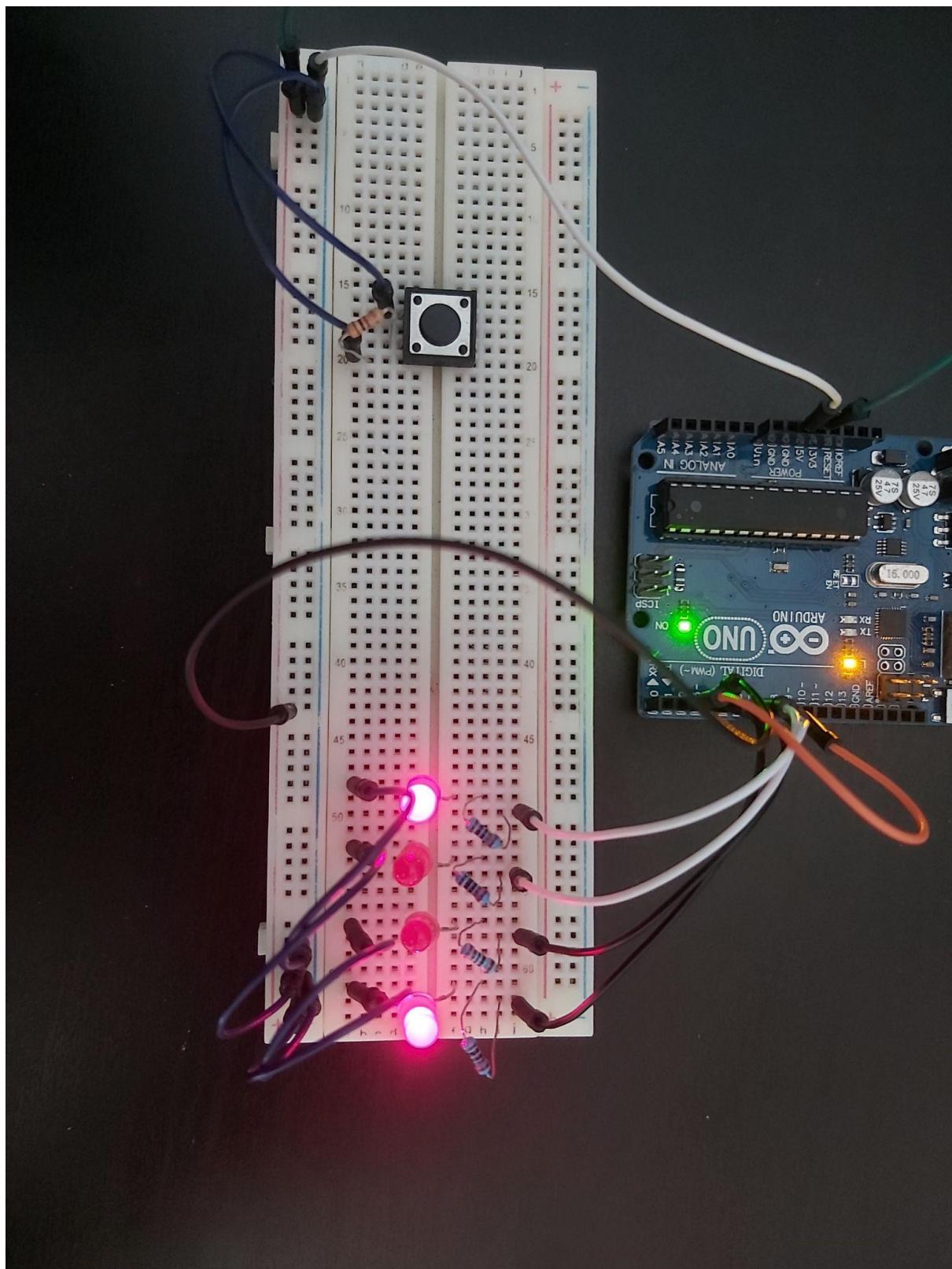


Figura 136 - Testes Contador Decrescente: S = 1001

4. Código

Exercício 1

```
#define pinA 2
#define pinB 3
#define pinF 4
#define pinS 5
#define pinL 6

boolean A, B, F, S, L;

boolean Led(boolean A, boolean B, boolean F, boolean S) {
    return S & !F & A | F & !A & !B | !S & F & !B | !S & F & !A | S & !F & B | !F & A & B;
}

void readInputs() {
    A = digitalRead(pinA);
    B = digitalRead(pinB);
    F = digitalRead(pinF);
    S = digitalRead(pinS);
}

void detectAnomaly() {
    L = Led(A, B, F, S);
}

void writeOutputs() {
    digitalWrite(pinL, L);
}

void setup() {
    pinMode(pinA, INPUT);
    pinMode(pinB, INPUT);
    pinMode(pinF, INPUT);
    pinMode(pinS, INPUT);
    pinMode(pinL, OUTPUT);
}

void loop() {
    readInputs();
    detectAnomaly();
    writeOutputs();
}
```

Exercício 2

```
#define pinSelector Ao

#define pinAo 2
#define pinA1 3
#define pinA2 4
#define pinBo 5
#define pinB1 6
#define pinB2 7
#define pinSo 8
#define pinS1 9
#define pinS2 10

#define pinL_CarryBorrow 11
#define pinL_Overflow 12
#define pinL_Zero 13

boolean Sel, A_o, A_1, A_2, B_o, B_1, B_2, S_o, S_1, S_2, CBw_o, CBw_1,
L_CyBw, L_Ov, L_Zero;

void calculateResult(boolean Sel, boolean An, boolean Bn, boolean CnBwnAnterior,
boolean * Sn, boolean * CyBw) {
    * Sn = CnBwnAnterior ^ An ^ Bn;
    * CyBw = (Sel ^ An) & Bn | CnBwnAnterior & (Sel ^ An) | CnBwnAnterior & Bn;
}

boolean flagOverflow() {
    return CBw_1 ^ L_CyBw;
}

boolean flagZero() {
    return !(S_o | S_1 | S_2);
}

void setFlags() {
    L_Ov = flagOverflow();
    L_Zero = flagZero();
}

void readInputs() {
    Sel = digitalRead(pinSelector);
    A_o = digitalRead(pinAo);
    A_1 = digitalRead(pinA1);
    A_2 = digitalRead(pinA2);
```

```

B_o = digitalRead(pinBo);
B_1 = digitalRead(pinB1);
B_2 = digitalRead(pinB2);
}

void writeOutputs() {
    digitalWrite(pinSo, S_o);
    digitalWrite(pinS1, S_1);
    digitalWrite(pinS2, S_2);
    digitalWrite(pinL_CarryBorrow, L_CyBw);
    digitalWrite(pinL_Overflow, L_Ov);
    digitalWrite(pinL_Zero, L_Zero);
}

void setup() {
    pinMode(pinSelector, INPUT);
    pinMode(pinAo, INPUT);
    pinMode(pinA1, INPUT);
    pinMode(pinA2, INPUT);
    pinMode(pinBo, INPUT);
    pinMode(pinB1, INPUT);
    pinMode(pinB2, INPUT);

    pinMode(pinSo, OUTPUT);
    pinMode(pinS1, OUTPUT);
    pinMode(pinS2, OUTPUT);
    pinMode(pinL_CarryBorrow, OUTPUT);
    pinMode(pinL_Overflow, OUTPUT);
    pinMode(pinL_Zero, OUTPUT);
}

void loop() {
    readInputs();

    calculateResult(Sel, A_o, B_o, o, & S_o, & CBw_o);
    calculateResult(Sel, A_1, B_1, CBw_o, & S_1, & CBw_1);
    calculateResult(Sel, A_2, B_2, CBw_1, & S_2, & L_CyBw);

    setFlags();
    writeOutputs();
}

```

Exercício 3

```
#define pinSel 3
```

```
#define pinSo 4
#define pinS1 5
#define pinS2 6
#define pinS3 7
```

```
bool Sel, Do, D1, So, S1, S2, S3;
volatile bool Qo, Q1;
```

```
unsigned long now, ago;
```

```
boolean flip_flop_D(boolean D) {
    return D;
}
```

```
void funcaoEstadoSeguinte() {
    D1 = Sel ^ Qo ^ Q1;
    Do = !Qo;
}
```

```
void funcaoSaida() {
    So = true;
    S1 = Qo;
    S2 = Q1;
    S3 = true;
}
```

```
void CLK() {
    now = millis();

    if (now - ago >= 200) {
        Qo = flip_flop_D(Do);
        Q1 = flip_flop_D(D1);
        ago = now;
    }
}
```

```
void clockGenerator(byte pino, float f) {
    static boolean state = LOW;
    static unsigned long t1, to;
    t1 = millis();
```

```

if (t1 - to >= 500. / f) {
    digitalWrite(pino, state = !state);
    to = t1;
}
}

void lerEntradas() {
    Sel = digitalRead(pinSel);
}

void escreverSaidas() {
    digitalWrite(pinSo, true);
    digitalWrite(pinS1, S1);
    digitalWrite(pinS2, S2);
    digitalWrite(pinS3, true);
}

void setup() {
    pinMode(pinSel, INPUT);
    pinMode(pinSo, OUTPUT);
    pinMode(pinS1, OUTPUT);
    pinMode(pinS2, OUTPUT);
    pinMode(pinS3, OUTPUT);
    pinMode(8, OUTPUT);

    attachInterrupt(0, CLK, RISING);
    interrupts();
}

void loop() {
    lerEntradas();
    funcaoEstadoSeguinte();
    funcaoSaida();
    escreverSaidas();
    clockGenerator(8, 1);
}

```

5. Conclusões

Com este trabalho finalizado, foi possível aprofundar os conhecimentos sobre o funcionamento de portas lógicas, o funcionamento do CLOCK e dos estados, tanto presente como seguinte. Mais concretamente, com o segundo exercício foi possível perceber qual o funcionamento de um erro nas contas de uma máquina calculadora. Para além disso, possibilitou a compreensão do conceito de memória e de como funciona um circuito sequencial.

6. Bibliografia

[1] Carlos Carvalho (2022). *Computação Física*.